

MDS

Adapter's Guide

MDS V6.1 Release
December 16, 2005

JPL D-33982

The information contained in this document has been designated by the California Institute of Technology (Caltech) as Technology and Software Publicly Available (TSPA). Copyright 2005. The copyrights and patents related to this technology are owned by Caltech. United States Government sponsorship acknowledged.

Table of Contents

1 Introduction.....	5
1.1 The Control Diamond.....	5
1.1.1 State Variable.....	6
1.1.2 Estimator.....	6
1.1.3 Controller.....	6
1.1.4 Hardware Adapter.....	6
1.2 Mission Planning and Execution (MPE).....	6
1.2.1 Time Points and Temporal Constraints.....	7
1.2.2 Goals.....	7
1.2.3 State Constraints.....	7
1.2.4 Goal Network.....	7
1.2.5 Elaborators and Elaboration.....	8
1.2.6 Scheduling and Promotion.....	8
1.2.7 Execution.....	10
1.2.8 Safing.....	11
1.2.9 Responsibilities of Planning and Execution Methods.....	12
2 Before You Start Coding.....	15
2.1 Prerequisites – State Analysis Artifacts.....	15
2.2 Package Structure and Namespaces.....	15
2.3 Make Rules and the MDS Build System.....	16
2.3.1 How Metamake Works and What it Does.....	17
2.4 Using the System ID Database.....	20
3 Derive a Software Component Architecture from State Analysis Artifacts.....	22
3.1 Measurements.....	25
3.2 Commands.....	26
3.3 State Values and State Functions.....	26
3.3.1 Value Histories.....	28
3.3.2 History Functors.....	29
3.4 State Constraints.....	29
3.5 State Variables.....	31

3.6 Implementing Planning and Execution Methods.....	32
<u>Order of Implementation.....</u>	<u>34</u>
3.7 Estimators and Controllers.....	36
3.8 Hardware Adapters.....	37
3.9 Goals, Elaborators, and Tactics.....	37
3.10 Data State Variables and Data Controllers.....	39
4 Instantiation and System Encapsulation.....	40
5 Runtime Scheduling and Execution.....	41
6 Adding Elf Events.....	43
7 Initialization.....	45
8 User Interface.....	45
9 Data Management and Transport.....	45
10 Porting to Physical Hardware.....	46
11 Optimization Notes.....	46
11.1 Memory Management.....	47
11.2 Optimization Strategies and Considerations.....	48
12 Testing.....	48
12.1 Writing Tests.....	49
<u>12.1.1 Unit Tests and System Tests.....</u>	<u>49</u>
12.2 MDS Test Harness and Test Frameworks.....	50
<u>12.2.1 Test Code Organization.....</u>	<u>51</u>
<u>12.2.2 CppUnit Unit Test Framework.....</u>	<u>51</u>
<u>12.2.3 Test Scripts and Script Helpers.....</u>	<u>51</u>
<u>12.2.4 Testing MDS Adaptations.....</u>	<u>53</u>
<u>12.2.4.1 Class API Unit Tests.....</u>	<u>53</u>
<u>12.2.4.2 Planning and Scheduling Tests.....</u>	<u>54</u>
<u>12.2.4.3 Execution Tests.....</u>	<u>54</u>
12.3 Debugging.....	55
12.4 Simulation.....	55
13 Advanced Topics.....	56
13.1 Multi-Threading.....	56
Appendix A. Open-Loop Temperature Controller Adaptation Example.....	57
Getting Started.....	57
A Tour of the Software Adaptation.....	59

Examining the Adapted Classes in More Detail.....	62
State Variables.....	63
Estimators and Controllers.....	64
Hardware Adapters.....	64
Data Classes: Measurements, Commands, and State Values.....	65
State Constraints.....	66
Goals and Elaborations.....	67
Unit Tests.....	68
Simulation.....	70
System Tests.....	71
Appendix B. Step-By-Step Process Used In the Adaptations.....	75
Glossary.....	83
Index.....	89

1 Introduction

This section provides a brief overview of the MDS control architecture as it applies to the software implementation of a control system. More information about how the architecture works, and how to use the features of the architecture to design a control system can be found in state analysis course material, packaged separately on the distribution CD, and in the papers described in references [1],[2],[3],[4],[5],[6],[7],[8],[9],[10], and [11]. These papers are also provided separately on the distribution CD.

1.1 The Control Diamond

The adapted parts of a control system that uses MDS are the elements that make up individual control loops: estimators, controllers, state variables, and hardware adapters. These are all described in detail in the state analysis course, and briefly repeated here for review, or for software developers who may not have been trained in state analysis.

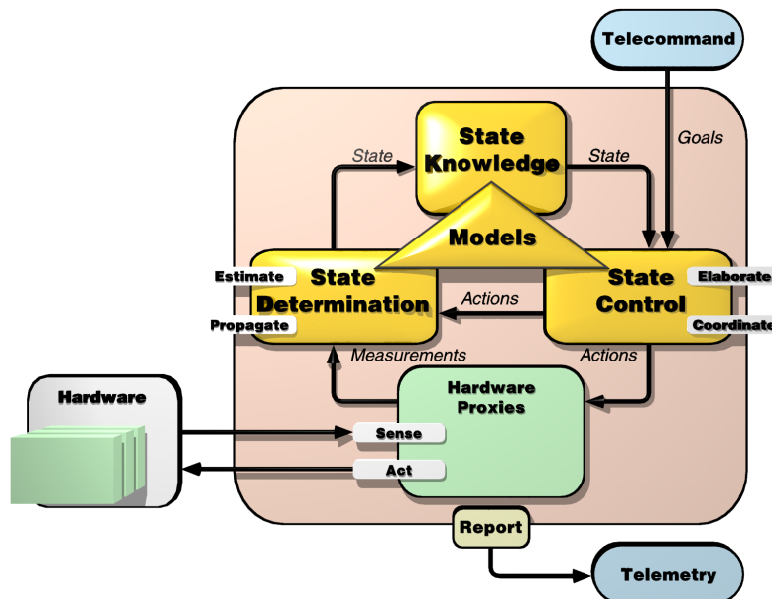


Illustration 1: MDS Control Diamond

1.1.1 State Variable

State variables represent the external physical states of the system to be controlled. They are containers for the state knowledge, in the form of state values or value functions, that explicitly describe the system states over time. State variables also provide standard methods that support the planning and execution of goals.

1.1.2 Estimator

Estimators are responsible for determining state knowledge from available evidence (in the form of measurements, or state values) and using that knowledge to update the state variable. The state variable is the sole repository for all state knowledge.

1.1.3 Controller

Controllers are responsible for performing any control actions needed to achieve physical control goals. These actions are usually in the form of issuing commands to an actuator through a hardware interface of some kind.

Estimators and controllers are collectively referred to as *Achievers*, because they cooperate to achieve all of the goals associated with their state variable.

1.1.4 Hardware Adapter

Hardware device command and measurement interfaces are abstracted through the use of hardware adapters. The primary responsibility of a hardware adapter is to provide some persistence for measurement and command values that pass through it so that these values can be queried asynchronously after having been produced or issued. In particular, it provides a history of commands so that estimators can use this as evidence in state determination.

1.2 Mission Planning and Execution (MPE)

This section describes the general theory behind the goal network and the interactions between the network and its elements. In particular, it describes how the internal functions interact with adapted classes such as state constraints and elaborators to realize a desired system control behavior.

1.2.1 Time Points and Temporal Constraints

A *Time Point* represents an event. A *Temporal Constraint* expresses a range of allowable durations between two Time Points. Temporal Constraints are the edges in the Temporal Constraint Network, where Time Points are the nodes. The specification of a Temporal Constraint includes a minimum duration, a maximum duration, a start Time Point, and an end Time Point. The event represented by a Time Point is said to occur when the Time Point “fires”. A Time Window is a range representing when a Time Point may fire. Temporal Propagation is the process by which a Time Window for each Time Point is derived from the Temporal Constraints in the Network.

1.2.2 Goals

A *Goal* is a constraint on state over a temporal interval. Goals express operator intent by specifying a State Variable, a State Constraint, a start Time Point, and an end Time Point. The duration of a Goal is determined by the Temporal Constraints relating its start and end Time Points.

1.2.3 State Constraints

A *State Constraint* expresses a set of allowable State Values. A Goal is said to have failed when the *Value History* of its specified State Variable fails to intersect the set expressed by the State Constraint.

The MDS 6.1 release implements state constraints as a distinct class from goals, although this is not an architectural requirement or mentioned in most of the state analysis process.

1.2.4 Goal Network

The data structure containing all of the goals, time points, and temporal constraints, is called a *goal network*, or *network* for short. A *proposed network* is constructed by making a copy of the currently executing network, and through the process of Elaboration, goals, temporal constraints, and time points may be added or removed, and the new network may be scheduled as described below.

Goal networks are portable data structures that can be exported through the process of *serialization*, and subsequently imported into the same or another deployment for use, thus allowing goal networks to be planned in one deployment and then shipped to another for execution.

1.2.5 Elaborators and Elaboration

Elaboration is the process by which a Goal is expanded into supporting sub-Goals. These sub-Goals or “child” Goals may themselves expand into other Goals thus creating a Goal hierarchy. The hierarchy is complete when all related state effects are represented. Goal Elaboration hierarchies are derived directly from the State Effects Diagram according to the guidelines specified in the State Analysis documentation.

An *Elaborator* is a supporting element of a Goal that is responsible for ensuring the success of its Goal by modifying the network, adding and/or removing sub-Goals, Time Points and Temporal Constraints, and providing a failure response when any of these sub-goals fails during execution.

Elaborators may have a set of one or more Tactics, which express alternate sets of subgoals that achieve equivalent results. When a goal network is being elaborated by the Elaboration Manager, it will try each tactic in turn until it finds a plan that is achievable, or it exhausts all possible combinations of tactics for all the given goals.

1.2.6 Scheduling and Promotion

Scheduling is the process by which a Goal Network is translated into *executable time lines*. An executable time line is made up of executable Goals, or *XGoals*. An XGoal is the product of merging one or more regular Goals. The Scheduler creates one XGoal time line for each State Variable. The container of all XGoal time lines is referred to as the goal network, or network. An overview of the Scheduling algorithm follows:

- 1) Order/sort the State Variables according to the State Effects Diagram, affecting State Variables before their affected State Variables. (see `SV::affectingStates` method, Projection)
- 2) For each State Variable, merge and order the Goals as described below to create an ordered time line of XGoals (see `Constraint::mergeFrom` method).

When Goals overlap temporally, and if they are compatible, the Scheduler will merge them into a single XGoal. If they are incompatible, the Scheduler will add Temporal Constraints which require the Goals to execute in series rather than in parallel. However, if these Scheduler-added Temporal Constraints are inconsistent with previously existing user-added Temporal Constraints, the scheduling process will fail (see Goal Scheduler

methods). The combined or merged constituent goals become the *ordinary constraint* of the XGoal.

3) Perform Projection.

Projection is the process by which State Effects are computed, recorded, and validated. An instance of a State Effect is called a projection. The projection effectively represents the sets of future state values in the form of state constraints. Representing projections as state constraints (also called the *projected constraint*) enables a direct comparison of the two, for the purpose of validating the plan against its predicted result. Specifically, if the state constraint specified by the Goal is a subset of (see `isSubsetOf`) the state constraint representing the projection, the plan is valid. The Scheduler performs Projection by querying adapter-implemented methods in State Variables and Achievers (see projection methods). The algorithms implemented by these methods are defined during State Analysis. The implementation of Projection methods should be straight-forward. If it is not, the State Analysis artifacts and/or process should be revisited.

Promotion is the process by which the currently executing network is replaced with a newly elaborated and scheduled network, called the proposed network. Any elaborator can request a proposed network at any time. A proposed network is essentially a scratch pad an elaborator can modify while the executing network continues to execute.

An overview of the proposal process follows:

- a) An elaborator “decides” (see Elaboration) it wants to modify the network.
- b) The elaborator requests a new proposed network.
- c) The proposed network is allocated and returned to the elaborator. The elaborator which initiates the proposal process is called the proposer.
- d) The proposer modifies the proposed network (see Elaboration) and instructs its children to do the same.
- e) When the proposer is satisfied with the proposed network it requests promotion.

An overview of the promotion process follows:

- a) If the proposed network has not been scheduled, schedule it.
- b) Verify no new Time Points in the proposed network are constrained to be in the past.

- c) Verify the achievability of transitioning from the XGoals in the old (executing) network to the XGoals in the new (proposed) network.
- d) Stop execution of the old (executing) network.
- e) Stop monitoring the XGoals in the old (executing) network for failure.
- f) Fire the promotion time point in the new network (thus, any goals that used this as their starting time point will become the current goals on their time lines).
- g) Dispatch the XGoals in the new (proposed) network to their State Variables.
- h) Start monitoring the XGoals in the new (proposed) network for failure.
- i) Start execution of the new network. (the proposed network becomes the executing network)
- j) Delete the old network.

1.2.7 Execution

This section describes the process by which an executable goal network is executed. Execution includes three distinct cooperating elements: time point firing, achiever behavior, and goal failure detection.

First, the Executive component in the MDS framework is responsible for propagating the temporal constraint network and evaluating goal transition conditions in order to fire time points.

A time point will fire when either:

- a) All of its outgoing XGoals are “ready to transition”. (see isReadyToTransition)

or

- b) Its Time Window expires (that is, its latest possible end time has passed)

whichever comes first.

When a Time Point fires:

- a) All of its outgoing XGoals are dispatched to their specified State Variables (see SV::startXGoal and Achiever::startXGoal methods).

- b) The XGoals Checker begins monitoring the XGoals for failure (see `SV::isStillSatisfiable` and `Achiever::isStillSatisfiable` methods).
- c) The XGoals Checker ceases to monitor the Time Point's incoming XGoals.
- d) The Temporal Constraint Network is re-propagated.

XGoals that are dispatched to their associated state variables are then delivered to each associated Achiever. It is the responsibility of those estimators and controllers to perform actions needed to achieve the given goals. Achievers can be periodic components that will continually operate to achieve their given goals. Different achievers can run at different rates, or even be passive – only performing actions when new goals arrive. Low-level active control loops live at this level.

The third part of the execution architecture is the executable goal checker, or `XGoalChecker`. It's job is to monitor the set of active executable goals (the ones currently in effect, and last delivered to each state variable) and to notice when goals fail. Checking is done first on the merged executable goals on each state time line by calling `isStillSatisfiable` periodically, or when the associated state variable reports a change in state. When this call returns false, the `XGoalChecker` calls `isStillSatisfiable` on each constituent goal of the failing merged executable goal. For each constituent goal where this returns false, a goal failure response begins, and proceeds as follows:

- a) its `getFailureResponse` method will be called. This in turn will call `getReElaboratableGoals` method on its parent Goal.
- b) the parent goal will call the `getReElaboratableGoals` method on its Elaborator to determine a response.
- c) the Elaborator may choose to respond in one of four ways. It can invoke a safing response by throwing an exception; it can cause the re-elaboration of its host Goal by placing that Goal in the list of Goals passed as an argument; it can propagate the failure upward to its parent Goal by calling `getFailureResponse` on its host Goal; or it can do nothing and effectively ignore the failure.

1.2.8 Safing

The MDS goal execution framework includes a mechanism for verifying that all goals continue to be satisfied over time. When something happens to cause a goal to no longer

be satisfiable a goal failure response is initiated as described in the previous section. The failure response allows the goal network, through its elaborators, to attempt recovery through the elaboration of alternate tactics. However, in the end, if no recovery is possible, the system provides a mechanism to promote a “safe” network that is supposed to constrain the system into a benign configuration that would allow for external diagnosis and recovery.

The planning and execution frameworks will automatically provide a default safe network, or you may install a custom network. The default safe network will place unconstrained goals on all state time lines. If controllers are designed to respond appropriately to unconstrained goals by placing the system in a benign state, then this response will be sufficient for most cases. For cases where a different response might be required for different situations (over time), the system allows a different safe network to be elaborated and installed in mostly the same way that ordinary goal networks are elaborated and scheduled, but with a few key differences.

A safe goal network must contain a sequence of goals that can be promoted at any time, and when the system is in any conceivable state. Since it will only be promoted when the system is in an unexpected failure state, it must be able to transition the system from that failure state into a safe state. And, since there can be only one safe network at a time, the safe network must be designed so as to be highly tolerant of the sorts of failures that would have caused the safe network to be promoted in the first place. Thus, the goals must be carefully designed to ensure that they will not fail, and can begin executing at any time and in any situation.

1.2.9 Responsibilities of Planning and Execution

Methods

Adapted control loop components interact with the MDS planning and execution frameworks (also called the Mission Planning and Execution, or MPE components) through a set of interface methods that the adapted classes implement. For the system to work as intended, each state variable and its associated achievers must implement these methods according to the rules given in this section.

Table 1.: Responsibilities of Planning and Execution methods

<i>method</i>	<i>Implemented in</i>	<i>Function</i>
Goal Scheduling Methods		
isAchievable (goal)	SV	Architecturally-defined method that would permit feasibility checking of individual goals. Not implemented in this release.
isAchievable (XGoal)	SV	Responsible for determining if a given merged goal is achievable. Used by the goal scheduler to reject schedules where the particular combination of goals on a given time line are incompatible. The default implementation returns true if the projected state constraint is a subset of the merged ordinary constraint (the merged goals). This implementation should only be overridden to add unusual other conditions.
IsTransitionAchievable (XGoal, XGoal)	SV, Achiever	Used by the scheduler to determine the feasibility of a transition between two goals.
mergeWith	Constraint	Computes the intersection of the sets of values allowed by two state constraints.
isSubsetOf	Constraint	Computes whether one constraint is a subset of the other. I.e., whether the set of state values defined by one constraint is a subset of the set of values defined by the other.
getProjectionType	SV	Used by the goal scheduler to determine which projection method applies to the given state. Must be one of the following: LOCAL - for states whose goal achievability depends only on the immediate goal SERIAL – for states whose goal achievability depends on the sequence of goals and an initial state. GLOBAL – for states whose goal achievability may depend on any other goals on its timeline.
projectLocally	SV	Models the predicted results of executing the given XGoal

<i>method</i>	<i>Implemented in</i>	<i>Function</i>
projectSerially	SV	Models the predicted results of executing the given XGoal sequence. Note that there are two distinct forms of this method, both of which must be implemented. The first takes an initial state value function and initial xgoal from which an initial projection must be determined. The other just takes subsequent xgoals, and projects state based on that goal only.
projectGlobally	SV	Project state based on the information available from the entire timeline.
<i>Goal Execution Methods</i>		
startXGoal	SV, Achiever	Delivers a new xgoal for achievement to an achiever by way of its SV.
isReadyToTransition	SV, Achiever	Given the current constraint and state value, and the pending subsequent xgoal, determine whether the system state would permit a transition to the new xgoal.
isStillSatisfiable	SV, Achiever	Assess whether the conditions needed to succeed in achieving the goal continue to be met. A false result initiates a fault handling process.

2 Before You Start Coding

In many systems the software developer role is distinct from the system engineering role. Developing requirements through the state analysis process is a system engineering task, but it is important that the developer be sufficiently familiar with the process to know how to interpret the resulting artifacts. The instructions provided in this section are primarily aimed at the software developers who will be interpreting the state analysis artifacts and translating them into executable code artifacts.

2.1 Prerequisites – State Analysis Artifacts

Before beginning any software development at least an initial state analysis of the target system should have been completed and produced the following artifacts which define the software requirements:

- List of states and state variables
- State effects diagram
- State models and representation rules
- Definition of goals and constraints
- Physical system model
- Measurement and Command interfaces and models
- Estimation and control algorithms
- Collaboration Diagrams
- Specification of mission planning and execution (MPE) methods (spreadsheet)
- Test case descriptions

2.2 Package Structure and Namespaces

MDS convention is to use package namespaces in C++ that reflect the directory structure relative to the source directory, similar to the Java language convention. This helps to make it easier to find the source code for a given class if you know its namespace. More importantly, the dependency analyzer will assume this naming pattern at least with regard to the location of header files in the export directory. You don't have to actually follow the convention in your adapted code, but you do have to define a package for each

directory you create, and use include directives that will include package headers from a directory path that is the same as the subdirectory path in the *source* directory tree.

2.3 Make Rules and the MDS Build System

The MDS build system is part of the MDS test harness. You don't have to use the MDS build system, but it probably provides the easiest way for you to build the MDS libraries, and it is easily extensible to work with added source trees. Thus, it should provide an easy default if you don't have some other system you need to use.

As part of the MDS test harness, the build system is invoked as one or more test scripts. The test harness controls the build and test environment, making it possible to build and test the system under multiple, parametrically-controlled configurations. The test harness also allows all regression tests including unit tests and system tests to express build prerequisites and to build their own executables as a prerequisite to execution. For convenience, a root Makefile is provided in the MDS_ROOT directory that can invoke the test harness to run named tests.

The delivery package includes two default test/build scripts which are invoked as the *metamake* and *core* targets in the root Makefile. For convenience, we'll use these make target names to describe the functions. The *metamake* test runs the MDS makefile generator and dependency checker. This tool is described in the next section. The *core* test builds all of the non-test target libraries defined in the source directory tree.

The MDS build system uses *metamake* to create one big Makefile for the entire system (the generated Makefile is the one written into the source directory – not the one in the MDS_ROOT directory. The one in the MDS_ROOT directory simply provides shortcut names for specific tests you run frequently). *Metamake* searches through the source tree looking for *make.cfg* files that express simplified build dependencies for each subdirectory. For example, it can define a target library, and which source files are members of that library. It can also express access control rules about which headers are intended to be public interfaces versus private implementation. *Metamake* uses this information and the source files for further dependency analysis to ensure that the make rules express a directed graph of dependencies between libraries and packages. Subsequently, build tests can invoke these make targets under different build configurations as defined by the test environment. This allows a library to be built using different compilers, or on or for different platforms using the same makefile, but with different test configurations.

2.3.1 How Metamake Works and What it Does

Metamake's make.cfg files follow the same syntax rules as the macro-definition portion of ordinary Makefiles. Anything after a # character to the end of a line is ignored, as are blank lines. Assignments are in the form of keyword = value <newline>, but only the predefined set of keywords can be used as described in table 1. Lines can be extended by escaping the newline with a backslash.

```
# Example make.cfg file
# comments start with #
# package name should match directory
# relative to 'source'
PACKAGE = Mds.Ra.Tc
# tell metamake to look in subdirs for more
# make.cfg files
SUBDIRS = *
# Build sources into library MyLib.a or .so
TARGET_LIB = MyLib
PROJFLAGS = SHARED # make that .so
SRCS = Example1.cpp Example2.cpp
GLOBAL_HDR = Example1.h \
             Example2.h
```

Example 1: make.cfg contents

Table 2: Directives used in make.cfg files

<i>Keyword</i>	<i>Value Definition</i>
PACKAGE	Name of the package namespace. This corresponds exactly to the java definition of package name. This defines where header files listed in GLOBAL_HDR will be exported to in the delivery directory. Can be

<i>Keyword</i>	<i>Value Definition</i>
	expressed in java style as “Mds.Fw.Init” or C++ directory style as “Mds/Fw/Init” or C++ namespace style as “Mds::Fw::Init”. Other code that includes exported headers must use fully-qualified includes: #include “Mds/Fw/Init/init.h”
TARGET_LIB or TARGET_TEST_LIB	Name of the target library to be built from the list of SRCS. None of the sources may contain a main function. The value is just a character string that will be used as the base name of the physical library file (excluding platform-dependent prefix or suffix – those will be added automatically by the build system).
TARGET_APP	Target executable application to be built from SRCS. In this case one of the SRCS must include a main function. A given make.cfg file may only define one TARGET_APP -or- TARGET*_LIB
SRCS	List of source files to be built as members of the named package and linked into the target library or application.
GLOBAL_HDR	List of header files in this directory that express public interfaces to the functions and classes defined in the sources in SRCS. These headers are exported into a subdirectory in the delivery directory where they can be included by other packages. Metamake also assumes that these headers express the interfaces to the package and target library defined for this subdirectory, so anything that includes these headers is assumed to have a dependency on the package and target library. Can include implementation files as well as header files for template bodies.
PRIVATE_HDR	List of header files in this directory that are private to the package and may only be included locally.

<i>Keyword</i>	<i>Value Definition</i>
	Metamake will not export these headers to where they can be included by other packages, but it will still compute dependencies on them.
PROJFLAGS	Override system defaults to declare build options for a library. Currently, this only controls whether a library will be built as an archive library or shared library. The default is “shared”.
SUBDIRS	List of named subdirectories to recurse into. Can include wildcards, or just * to recurse into all subdirs. Note that directory recursion only happens when Metamake runs – not when make actually runs.
UTESTSRCS	List of unit test sources. Each listed source file should contain a main function that implicitly depends on the TARGET_LIB. The main function performs some set of unit tests on the target library functions and returns the number of errors.

Table 3: Common metamake errors and corrective actions

<i>Error</i>	<i>Description</i>
Cyclic dependencies between target libraries	The metamake output will try to describe the cyclic dependency as expressed by one or more includes. You cannot define target libraries with cyclic dependencies because many linkers will have trouble with this. You may need to refactor certain libraries to restructure the dependencies.
Target library dependency on test code	Metamake will allow code defined in a TARGET_TEST_LIB to make calls into a TARGET_LIB. Test code can do whatever

<i>Error</i>	<i>Description</i>
	it needs to perform a given test. However, core library code may not express dependencies on anything but other target library code in order to avoid uncontrolled dependencies on code that is not part of the target delivery.
Missing source files or headers	Metamake verifies completeness and consistency of includes. The most common reason for it not being able to resolve an include is that the target header file was not exported in a GLOBAL_HDR specification, or there is a mismatch between the PACKAGE name under which it was exported, and the path used to include it. Metamake will also report as errors files that are listed in make.cfg but don't exist in the directory.

2.4 Using the System ID Database

Some of the MDS frameworks depend on having assigned numerical identifiers. For example, state constraints, measurements, and state values are all defined as subclasses of polymorphic base classes. Instances of these classes will need to be identified when passed through a base-class reference in certain cases. Although this is generally what the C++ typeid system was designed for, the typeid system was not meant to be portable across compilers and deployments. Since many of these data classes do need to be transported (as telemetry and goal networks) between deployments, MDS provides its own system for assigning numerical identifiers that can ensure consistency across compilers, platforms, and deployments. It does this using an external database to manage the numerical assignments and a software framework that allows the assigned values to be imported as named enumerated values into the source code at compile time.

The System ID Database framework (Mds::Fw::Sid) provides an interface between the external configuration control database and the source code. You don't have to have an external database. If you did, you would set it up to export its contents in the form of a table expressed as comma-separated text. However, a default version of this table is provided in the framework and can be edited by hand to add new entries.

The database table defines five columns: instance name, assigned id number, namespace or group name, polymorphic flag, and description.

Table 4: Sid Identifier Database Entry Description

<i>Column Name</i>	<i>Description</i>
Instance Name	Namespace-qualified name of the class or object instance. In the generated Database.h file that Sid will produce this name will be converted into an enumerated value name with all the periods converted to underbars. For example, "Mds.Fw.Elf.SimpleEvent" becomes Mds::Fw::Sid::Database::Mds_Fw_Elf_SimpleEvent.
Assigned numerical ID	Assigned id number (int). Must be unique within namespace
Namespace or Group Name	Name of a namespace or group. This can be the base class of a polymorphic type or a type for which instances are to be assigned.
Polymorphic (boolean Y or N)	Not used in the code right now, but intended to distinguish between namespaces that assign values to object instances versus subclass types.
Description	Quoted text description or notes associated with the entry. Not used in the code.

3 Derive a Software Component Architecture from State Analysis Artifacts

- A reference adaptation is provided in the distribution source tree that includes a working temperature controller. The process described here will refer to these examples extensively. The example source code can be found at `source/Mds/Ra/SimpleThermostat`. The requirements developed for this adaptation can be found at `docs/example/tcanalysis.pdf` and `docs/example/methods.xls`.
- In the MDS source tree you can find some example adaptation classes that can be used as adaptation templates. In `source/Mds/Ra/Example` there are generic, unadapted, examples of most of the classes you will need to adapt to implement a control loop. These are heavily commented to explain the methods you need to implement.

There is no single way to approach the task of translating the state analysis artifacts into code. However, it seems reasonable to start with the simpler elements and work up to the more complex ones. The following sections describe each of the adapted elements of a control system starting with what are probably the simplest and most independent classes and working up to the more complex, and more interdependent ones.

By way of advice, it is good practice to design an implementation plan based on an increasingly-complex sequence of test cases that verify increasingly integrated capabilities. Using this approach you will likely only implement some of the more primitive methods in the first pass through this list, and add higher-level functionality in subsequent iterations.

A simplified development and test plan might go something like this:

1. Define the classes you're SURE you're going to need based on the state analysis requirements. Determine dependencies on other packages and generally start with leaf nodes in the dependency graph. Nail down package and class names before you start. Create the new subdirectory in the source tree for your new package (start a new project under source and sketch out the package/directory organization to a first order before drilling down much farther). Populate the directory tree with make.cfg files that simply declare the subdirectories (SUBDIRS = *).
2. Start by implementing Measurement and Command classes as described in sections 3.1 and 3.2, below, using the provided examples as templates. Use find & replace to edit the class and namespace names, but don't add any class attributes yet. You will need to assign class ID values in the Sid database for these classes.
3. Edit the make.cfg file in your code subdirectory to define the PACKAGE, TARGET_LIB, SRCS, and GLOBAL_HDR entries. Run metamake (make metamake in the root directory). Check the test output in results/tmp.metamake if the test failed. Once metamake passes, build the new sources by running *make core* in the root directory. Again, check for errors. Iterate until you get the generic classes to compile. Now you can flesh them out to a first order and get those changes to compile.
4. Next you should start thinking about unit tests. It is very useful to define unit tests for individual data classes such as we have just defined. Unit tests can verify that these classes implement their constructor and accessor methods correctly, and will be very useful for testing constraint merge methods later on (these are rather more difficult to test in operational system tests). You can write unit tests in the form of source files with a main function that instantiates your Measurement class and exercises its API. You can use *CppUnit* frameworks in these unit tests if you like. *CppUnit* provides a nice way to account for test results, but using an ordinary assert will also work to check test conditions. Add the unit test sources to a UTESTSRCS assignment in the *make.cfg* file. Then write a little test script to exercise these unit tests (see section 9 for more information about how to write tests in the MDS test harness).

5. Implement state values and state functions next. While you're at it define the state history type. [See 3.4.x, below]
6. Implement the SV next. [See 3.4, below]
7. Implement the hardware adapter next, as described in section 3.7. There isn't a prototype for this because every hardware adapter is going to have different interfaces. Implement methods for commanding and querying measurements as needed. If you need to implement a simulation model in the hardware adapter, do it now, and write a unit test to verify the simulation.
8. Sketch out the Estimator and/or Controller as described in sections 3.5 and 3.6, below, next.
9. Integrate the control loop. You may want to define container classes that aggregate the whole control loop and hook up the components. Make sure the SV gets an assigned instance ID. Write a unit test to run the control loop in its unconstrained state. At this point you will need to implement a simple rate group scheduler in your test to run the components. See sections 4 and 5 for more details.
10. Sketch out the state constraint as described in section 3.8, below. You might want to start out by just implementing a knowledge constraint. It is also recommended that you implement a unit test for the constraint class that exercises its merge and subset methods. This can save a lot of time later, because a lot of subsequent functionality depends on these functions working correctly. Then update your test to schedule and execute a knowledge goal. Make sure the estimator actually updates the state variable's history correctly. You'll need to implement the goal scheduling methods in your SV and (optionally) achievers. Once you get a basic test working you can flesh out the constraint and the related scheduling and execution methods in the SV and achievers. It is generally simpler to test at this level using simple terminal goals containing no elaborators (you can just construct these goals in the test) and manually constructed, simple goal networks.
11. If the package requires an Elaborator or a customized Goal you can start on that after everything else is working. The details of this step are described in section 3.9, below. In general, you want to have some confidence that the subgoals you're going to schedule as Tactics have all been verified individually as much as possible before you start combining them.

12. Iterate steps 2 through 11 for state variables working up from the leaf nodes in the state affects graph. Add new system tests as control loops are closed, state affects are elaborated, or additional complexity is integrated.

3.1 Measurements

A Measurement is a portable time-tagged wrapper for a piece of evidence sampled from hardware. The wrapper class provides a consistent polymorphic interface with architecturally-defined semantics: mainly that measurements are consistently time-tagged. The evidence contained within a measurement is a sample of data as defined by the target hardware.

Adapting from the example: Most of the classes described here can be adapted from the example prototypes provided in `source/Mds/Ra/Example`. These examples are heavily commented to describe the methods and attributes you need to adapt. In general, you can just copy the header (.h) and body (.cpp) files into your source directory and make the following changes:

- Change the namespace. You may need to add contained namespace declarations to do this.
- Global find and replace to change the class name to your particular class name (if necessary). Frequently it will be safe to use the existing class name as long as it is unique in your namespace.
- For `PolySerializable` classes you'll need to register an assigned class identifier in the `Mds::Fw::Sid` database. See section 2.5. Update your cpp file to use the assigned ID enumerator.

The only thing you should need to add to the Measurement example class are as follows:

- Value representation: This is the measurement's actual payload data, and it is usually defined by the underlying hardware from which it is obtained. Values as small as a single bit (boolean) can be represented, though it is important to take

into consideration the fact that compilers will normally use at least full word of memory to store a given value. For most measurements, this footprint expansion is acceptable as long as they remain in memory in small quantities. If you need to have many instances of measurements that store boolean values you might want to consider consolidating measurements or hardware adapters (arrays of switch positions come to mind).

- Specialized constructors. Add one or more constructors that allow an instance of your measurement class to be created given the payload data and a time value. Be sure to call the appropriate base class constructors to pass the time tag.
- Serialization methods. The *writeObject* method and its complementary deserializing constructor must be implemented to enable these measurement objects to be stored and transported as telemetry data. These methods can use functions in the *Mds::Fw::Ser::ObjectInputStream* and *Mds::Fw::Ser::ObjectOutputStream* classes to read and write the specialized attributes.
- Your measurement class should provide const accessor methods for any payload data values it contains. Mutator methods are generally not provided, since measurement objects should be immutable once constructed.

3.2 Commands

Commands are messages from a Controller to a Hardware Adapter directing some action. Their content depends on the hardware. As with Measurements, the MDS class interface provides a standardized wrapper that provides a time tag and some standard interface methods. The steps involved in implementing a command are identical to those needed to implement a measurement.

3.3 State Values and State Functions

State variables should be able to answer questions about their values continuously at any required point in time, or over any required interval. The specific representation requirement will derive from the estimation and control algorithms that derive from a given state. Frequently, it is sufficient for a state variable to provide a simple constant state function that is continuous in time, but possibly discontinuous in value. As long as the values are updated frequently enough to keep the value deltas small,

The state value representation(s) should generally be defined by the state analysis requirements.

One implementation consideration is the representation of unknown values. All state variables must be able reflect unknown state values. The simplest way to do this is to use a state value with a boolean “unknown” flag. Another way is to define a value domain that is a subset of what the primitive value types can represent, and interpret any value outside that limited domain as unknown. The boolean flag option uses more memory per object instance, but the alternative requires more code to interpret unknown values. You could also use an explicit “unknown” state value class. These optimization choices should be driven by the system requirements for how these values will be used and accumulated in the running control system – i.e., whether you have to optimize for size or speed, or both.

For states that change slowly, or are estimated frequently enough, a constant state function provides a state representation that is continuous over time, and sufficiently continuous in the value domain for many closed-loop control problems. Constant state functions also work well for enumerated states. Because this is a common design pattern, the MDS state frameworks provide a template for it in the ConstantStateFunction. ConstantStateFunction requires one template parameter that indirectly defines the class traits. The traits template parameter names a class or struct that defines three types: first a typedef alias that identifies the target StateValue class as StateValueType; second, a typedef that identifies the numerical identifier type; and third an enum that assigns m_class_id to the assigned class id value in the Sid database.

```
struct MyStateFunctionTraits {
    typedef MyStateVal StateValueType;
    typedef Mds::Fw::Sa::State::StateValue::IdType IdType;
    enum { m_class_id =
        Mds::Fw::Sid::Database::ASSIGNED_VALUE };
};
typedef ConstantstateFunction<MyStateFunctionTraits>
    MyStateFunctionType;
```

Text 1: Example of ConstantStateFunction traits declaration

For states that require more continuity in the value domain, other state functions such as polynomials can be used for scalar states.

3.3.1 Value Histories

State Variables representing estimated states always have at least one value history. Other components can also have value histories. Hardware Adapters can have histories of their commands and measurements, for example, and other components can produce *sample histories* for telemetry purposes. The basic process is the same in all cases except that the state variable base class (SV) provides a container for its value histories, where other base classes do not. If you want to use a value history in some other class you will need to define it and instantiate in your adaptation.

A value history can be any container, including a primitive variable. In such a simple example the history can only represent a history of one value, but that may be all you need in some cases. State value histories need to meet the requirements for representing state values, including the requirements for time labeling, so state value histories are usually somewhat more complicated than just primitive values.

The depth of a value history and its semantics depend on the requirements of the control loop in which it lives and the kinds of goals that will be applied. Certain goals or control algorithms require more than just the last sample. For example, an algorithm may depend on the rate of change of a state (which typically requires the last two values to compute), or the state relative to an earlier state.

Value history containers can serve as synchronization points between different threads of execution. In a state variable, for example, the actual state is in the value history. A state variable will typically have an estimator calling to update its state, perhaps a controller calling to get current state, and other estimators calling to get evidence for their estimation. Plus, the XGoalChecker and ElaborationManager may be calling for queries having to do with current execution and planning. Each of these calls could potentially be done on a separate concurrent thread of execution, so the value history provides a convenient place to synchronize them.

Synchronization is not automatically provided in the base class because some adaptations may choose not to pay the overhead of using multiple threads. It's also possible to schedule all of these components to run on a single thread. The example value history

containers provided in the framework demonstrate the use of atomic operations to provide thread safe behavior instead of relying on a mutex or semaphore. This can be significantly faster for simple histories. Note that if you need to define a deep history container that supports any operation that needs to iterate the container, then you probably cannot avoid using a mutex to prevent it from changing while you iterate.

3.3.2 History Functors

State variable value histories are initialized, queried, and updated through a set of base class interfaces that use specialized function objects, or *functors*, to do the work. These functors are typically customized for the particular value history container and content value types.

Query Functors. Query functors can be adapted to provide different ways to access a complex state history, and they can provide different accessor methods to manipulate the returned values. They can also be optimized to take into account the behavior of the value history they will associate with.

Update Functors provide methods to update the state and enforce the update rules and semantics for the particular value history.

Init Functors are specialized update functors that assume the previous state is uninitialized. These functors may actually be used to construct the value history.

3.4 State Constraints

First, a bit of translation. The state analysis course, and derived guidelines will generally talk about goals and state constraints as synonymous, but the current set of MDS frameworks provide distinct classes to represent these concepts. In the frameworks a goal is a container that associates a state constraint with two bounding time points, so Goal and state Constraint are two separate classes. When you see requirements referring to merging and subsetting goals, interpret those as referring specifically to the state constraints that implement those goals. When you see requirements related to the elaboration of a goal, those will be implemented in the elaborators and tactics described in a later section.

Think of state constraints as representing sets of possible state values. Though the name may sound like an oxymoron, the “unconstrained” constraint simply represents the set of all possible state values for a given state. Subsets of this set represent other possible values or value ranges required of the given state. State constraints can also be

“maintenance” or “transition” constraints for some states. That is, goals that require a state to be held within a given range may require some additional work ahead of time to transition the state into the required range. Not all states require transitions, so the maintain/transition flag is generally something you have to implement in your adapted constraint. However, the “unconstrained” constraint is implemented in the base class.

The most important methods in state constraints are the constructors, and the mergeFrom and isSubsetOf methods. The constructors need to provide signatures that allow Elaborators to efficiently construct the kinds of constraints needed to define required goals. The mergeFrom and isSubsetOf methods implement simple mathematical set operations. The mergeFrom method finds the intersection of the object constraint and the argument constraint. That is, it finds the intersection of the two sets of allowable values on the associated state. The isSubsetOf method determines whether the object constraint set is a subset of the argument constraint set. Note that since the “unconstrained” constraint is defined as the set of all possible states values, any physical constraint is always going to be a subset of this set.

Knowledge constraints are usually implemented as additional attributes of a control constraint. In general, some knowledge is required in order to achieve any control constraint. The examples demonstrate a form of modal estimation where state estimates are only updated when a knowledge constraint is being achieved. The state constraints for these states define KNOWN as a subset of unconstrained, and all control constraints are a subset of KNOWN. One way to do this is to implement a Constraint with an enumerated mode, where the mode has enumerated values for KNOWN, and any other kinds of physical constraints needed. All higher enumerations imply a knowledge constraint, but the KNOWN value allows for a constraint that constrains knowledge, but not the physical state. This is useful for cases where the state is required to estimate another state. This pattern can work even when the constraint needs to have attributes describing the quantity or quality of knowledge.

The implementations of the merge and subset methods for discrete states are usually straightforward. Constraints on scalar states may be efficiently represented as ranges. The example demonstrates the use of a math framework class to support range operations.

Constraints should provide a default constructor that creates an “unconstrained” constraint. This is only a convention rather than a requirement, but all of the examples assume this convention.

It is highly recommended that separate unit tests be written for each constraint class in order to exercise and verify the merge and subset methods (particularly all combinations that will show up in system tests). Testing these methods in a system test context with other states can be very tricky because so many other things are happening at the same time, and because it is often impossible to schedule individual goals by themselves (because of state effects and dependencies). The canonical nature of these classes and their key methods should make them amenable to automated test generation techniques and automated coding techniques. This is a key objective of future work.

3.5 State Variables

For each State Variable, the process of state analysis should have determined the following attributes:

- Is the state value to be estimated, or derived? Estimated states must have a value history, and must support the update interface. Derived states require adapted query methods that will synchronously query source state variables in order to derive a new state value.
- If the state is estimated, what are the bounds of knowledge and data state requirements that determine how deep or detailed the state value history must be?
- Is the state affected by other states, or does it affect other states? State effects are configured at system construction time using the method `SV::affects(SV* other)`.
- How is the state value projected into the future for planning purposes? The base class default is to perform a LOCAL projection by simply projecting the ordinary constraint forward. Override `getProjectionType()` and the appropriate projection method (`projectLocally`, `projectSerially`, or `projectGlobally`) if the given state uses SERIAL or GLOBAL projection, or if you need a special implementation of LOCAL projection (unlikely). You only need to implement the one projection method applicable to the form of projection `getProjectionType` returns.
- Are there requirements or constraints on how the planning and execution methods must be implemented? The SV base class provides default implementations of all of the planning and execution methods. Most of these implementations attempt to call methods with similar signatures on the associated Achievers. For example, the

default implementation of `isAchievable` calls a method with the same signature on each associated Achiever (associated by calling `linkAchiever`). This allows each achiever to weigh in on the achievability of its particular part of each given goal.

- Other methods like `isStillSatisfiable` only provide a trivial default implementation, so should always be overridden in an adapted SV.
- What are the concurrency requirements on this state variable, and on the architecture in general? Implement synchronization in the value history as necessary.
- State effects relationships are implemented using the `SV::affects()` method at the time the system architecture is instantiated.

Notification: MPE depends on implementation of an abstract notification interface in the SV framework. A fast and simple implementation of this notification interface is provided in the `Mds::Ra::Mpe` package. This package implements a notifying SV and a fast wait-free signaling queue in the form of a bit vector. If you choose to use this you can simply derive your state variables from `NotifyingSV` rather than the SV base class, and instantiate the `BitQueue` somewhere.

3.6 Implementing Planning and Execution Methods

The planning and execution methods refer to the set of methods on state variables, and their achievers, by the MDS goal planning and execution frameworks to support their runtime operations. The state analysis course describes the design and architectural requirements of each of these methods, along with guidelines for establishing their functional requirements. The specific state analysis of a given system should thus define specific functional requirements for each of these methods. However, some translation may be needed to get those functions efficiently into code.

General notes:

- The goal network can represent unconstrained constraints in two distinct ways: either as an actual instance of a constraint object whose `isUnconstrained()` method returns true, or as a null pointer. Methods that extract constraints from Goals and XGoals should take this into account.

- Any method that uses state constraints will need to determine the type of constraint and downcast the received object to its specific type for use. The simplest and most common case is for a given state variable to recognize just a single type of constraint. For this case the constraint base class provides a templated downcast method that will perform a static cast to the given type (with an assertion for debugging). If this is not the case, then you will need to use a switch statement whose conditional calls `getType( )` on the constraint object, and whose cases downcast the object to the particular constraint types.
- The goal Network class allows, as an optimization, XGoals to be constructed with NULL constraints representing the unconstrained constraint. Thus, methods that retrieve the ordinary or projected constraint from an XGoal should interpret a null pointer as an unconstrained constraint. This test should be done prior to any attempt to downcast the pointer, because the downcast will fail if the pointer is null.
- The default base-class implementations of all of these methods will return true, which will generally be the most optimistic and permissive case. One strategy for implementing these methods is to start out with an adapted state variable (SV) that
- The `isTransitionAchievable` methods may assume that the constraints in each of the given XGoals has already been tested for individual achievability, so it only needs to verify whether or not the given transition is allowed.
- When implementing projection rules from requirements specified in the spreadsheet form, you should be aware that you have to combine the results of projected control and knowledge goals to arrive at the actual constraint to be created (the requirements may mention the need to perform a symbolic AND). For example, the projection for a health state may specify a control part and a knowledge part separately. The control part will likely say that when the state is unknown the projected constraint is unconstrained. However, the projected knowledge goal may still be KNOWN. The actual projected constraint in this case should be whatever form of the constraint represents this combination.
- Note that much of the logic needed to implement the `state constraint mergeFrom` method will normally reside in its `isSubset` method. Consider implementing one in terms of the other.

Order of Implementation

To simplify testing, the following order of implementation is suggested. This procedure assumes that you start out with default implementations of the SV and Achiever methods that return true, and implement the methods one at a time in the following order:

- State constraint methods `mergeWith` and `isSubsetOf`. If you implemented these methods as part of the earlier procedure for implementing state constraints, and tested them in a unit test, all of the subsequent steps will be greatly simplified. This is because the planning and execution methods depend heavily on these supporting methods, and errors in the supporting methods may be difficult to detect when exercised in a much larger and more complex system test.
- State variable projection methods
- `isAchievable` and `isTransitionAchievable` (these depend on projections). Verify with scheduling tests.
- `IsReadyToTransition`. Verify with scheduling and execution tests.
- `IsStillSatisfiable`. Verify with goal failure tests.

Table 5: Planning and execution method notes

<i>Method Name</i>	<i>Implemented In</i>	<i>Description</i>
<code>getProjectionType()</code>	SV	Return LOCAL, SERIAL, or GLOBAL to reflect which projection method will be implemented. Default: LOCAL
<code>projectLocally(xgoal)</code> <code>projectSerially(xgoal)</code> <code>projectSerially(xgoal, statefunction)</code> <code>projectGlobally(xgoal)</code>	SV	Default implementation of these methods simply uses the ordinary constraint as the projected constraint on each XGoal in the timeline. Note that for SERIAL projection there are two method signatures that you should implement.

<i>Method Name</i>	<i>Implemented In</i>	<i>Description</i>
<code>isAchievable(xgoal)</code>	SV	Default implementation in SV will return the result of AND-ing the results of calling the same signature on each associated Achiever. NOTE: the requirements derived from state analysis assume that this methods doesn't even exist, and that achievability will always be determined by checking to see that the xgoal's projected constraint is a subset of the ordinary constraint. See example, below.
<code>isTransitionAchievable</code>	SV (or Achiever)	Default implementation in SV will return the result of AND-ing the results of calling the same signature on each associated Achiever.
<code>isAchievable(goal)</code>	SV (or achiever)	NOT IMPLEMENTED in this release.
<code>IsReadyToTransition(xgoal,xgoal)</code>	SV (or Achiever)	
<code>IsStillSatisfiable(xgoal)</code>	SV (or Achiever)	Default SV implementation will call a similar signature on each achiever and return the result of applying logical AND to each of these results. Default achiever result is true.

<i>Method Name</i>	<i>Implemented In</i>	<i>Description</i>
IsStillSatisfiable (goal)	SV (or Achiever)	Default SV implementation will call a similar signature on each achiever and return the result of applying logical AND to each of these results. Default achiever result is true. The logic for the two isStillSatisfiable methods should always be the same, so you should implement these using a common method that takes a state Constraint as argument.
startXGoal	Achiever (or SV)	Required by Achiever interface even if the achiever does nothing. An SV that has no achievers may need to override this method in order to save a reference to the currently executing xgoal in order to implement isReadyToTransition.

3.7 Estimators and Controllers

Estimators and Controllers are forms of Achievers associated with a state variable. The MDS architecture stipulates that there can be no more than one estimator and one controller for each state, but there can be fewer. Derived states might have no estimator, and there can be cases where one estimator updates multiple state variables. Similarly, there can be cases where states are not directly controllable, and so have no controller, or where one controller controls multiple states.

The MDS framework for Achievers is very simple. The base class `Mds::Fw::Sa::State::Achiever` defines some virtual planning and execution methods that state variables prefer to delegate to achievers. Logic related to the scheduling and execution of knowledge goals is usually delegated to an estimator, while logic related to

the scheduling and execution of control goals goes to a controller. Requirements defined through the state analysis process should specify all this.

In addition, estimators are usually given the responsibility for initializing the state of the state variable they update. This is because the state variable is a passive component, while the estimator is likely to be scheduled for periodic execution.

3.8 Hardware Adapters

Unlike many of the other adaptation classes, there is no standard base class for hardware adapters. A hardware adapter's interface will depend on whether it is commandable (an actuator of some kind), whether it produces measurements (a sensor of some kind), and whether it needs its own cpu cycles to do work beyond what it can do on the command and measurement interfaces.

Hardware adapters should generally be passive components (no run method), like device drivers. However, it may be convenient to embed a simulation model in a hardware adapter for testing purposes, and in that case it is often useful to do the simulation work on a separate run call to keep that work distinct from the control and estimation work.

3.9 Goals, Elaborators, and Tactics

Illustration 2 depicts the relationships between Goals, Elaborators, and Tactics. Adapted goal classes serve to statically associate a particular constraint or kind of constraint with a particular state variable (SV), and optionally, an Elaborator. Typically, a distinct elaborator class is adapted for each unique Goal class. An ordinary goal will associate with one state variable, one constraint, and one elaborator. A *terminal goal* will have no elaborator, because it doesn't need to elaborate any subgoals to be achievable.

A *macro goal* is a goal that doesn't associate with a state variable, and thus cannot have a state constraint either. A macro goal provides a mechanism by which entire activities can be elaborated onto multiple states in a single coordinated operation. So, a macro goal must always have an elaborator with at least one tactic.

Adapted Goal classes can be defined as long as they only specialize a constructor or methods that will be called from the constructor, nor may they have any specialized attributes. The current network serialization design only serializes the contents of the base class.

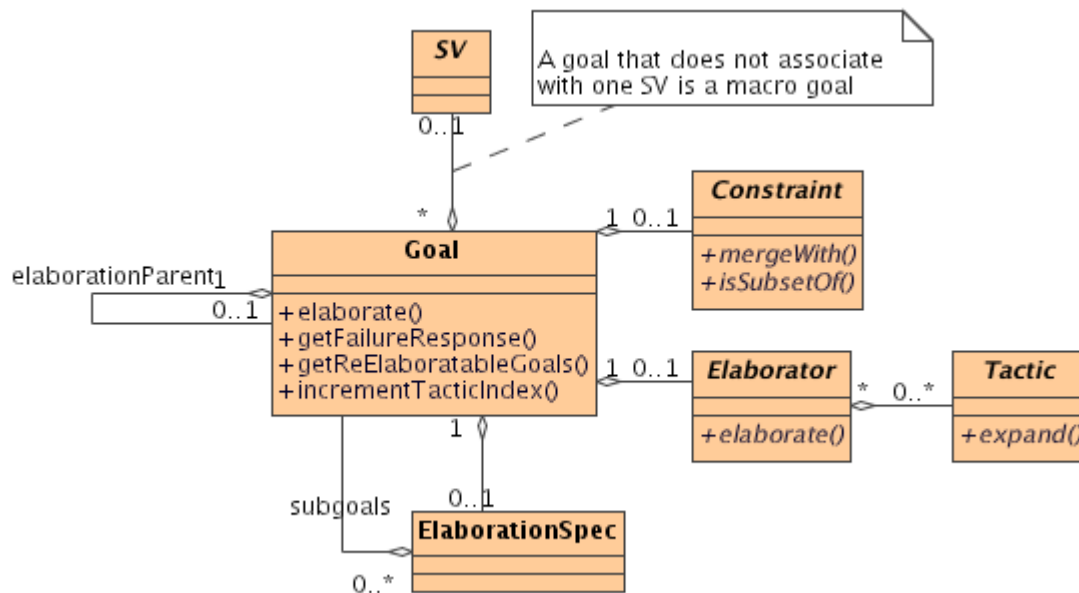


Illustration 2.: Goal class relationships with other adapted classes

Elaborators may be fully polymorphic.

Every elaboratable Goal must have an Elaborator class defined for it. An Elaborator may define its tactics as separate Tactic classes, but it doesn't have to. If you choose to implement the tactics inline in the Elaborator, be sure to also override the `getNumberOfTactics` method to reflect the number of tactics available.

The main things the Elaborator needs to implement are the `elaborate` method, and the `getReElaboratableGoals` method. The `elaborate` method is responsible for expanding the requested tactic. This is done by constructing a new `ElaborationSpec` container, and adding any new Goals, time points, or temporal constraints to this container, and returning the container. Canonically, this is done in separate Tactic classes.

Implementation Notes:

- Elaborators and Tactics should not add any element to the `ElaborationSpec` container that it doesn't create internally. Specifically, it should not add the parent time points or Goal (or itself) to its `ElaborationSpec`. During the process of re-elaboration, a Goal's previous elaboration as specified in its `ElaborationSpec` is removed (and possibly deleted) before the next tactic is elaborated. If the Goal's `ElaborationSpec` contains itself or its bounding time points, then memory corruption will occur.

- Goals that have multiple tactics that can be re-elaborated as a fault response need to avoid elaborating subgoals against the starting time point associated with the parent goal. If these tactics had to be re-elaborated during execution it is likely that the starting time point will have already fired, and it is not possible to schedule new goals against a time point in the past (you can't change history). A recommended technique is to create a new starting time point for the activity within the tactic expansion, or use the promotion time point. Then, schedule new goals against the new time point, and add a temporal constraint tying the new time point to the original starting time point. This is an example of conditional elaboration.

3.10 Data State Variables and Data Controllers

The design intent is that each value history container type should also associate with a data controller and data state which will manage the data in the history container, moving older data to data products for persistent storage and transport. Controlling the movement of data between the in-memory data cache and its extension in deep, persistent storage is performed through the use of *data goals*, data states, and data controllers.

A data goal is like a knowledge goal except that it constrains the state of a value history's content after the fact. That is, an estimator is the achiever responsible for updating a state variable's value history, and so estimators are primarily responsible for creating the data content in a value history. A data controller is an achiever responsible for what subsequently happens to the data. A data controller is an achiever for a data state which describes the condition of a value history. Since the value history is a part of the control system itself, data states are defined as a special case of a derived state variable that can have no value history or estimator of their own. Their state value is the physical state of the value history.

The value history to data management interface is not implemented in the MDS 6.1 release, and the current examples do not include any examples of data states.

4 Instantiation and System Encapsulation

This section talks about instantiating the components we've defined as classes in the previous sections.

Every control system will need to have a place where the state variables, controllers, estimators, and hardware adapters are all instantiated, connected, and initialized. This is also referred to as the architecture instantiation. It is convenient to define one or more container classes for this purpose. In a system consisting of more than a handful of components it is often useful to organize the components into groups according to either hardware relationships, or levels of interaction with one another, and to define separate subsystem containers for each. This is particularly true for repeating subsystems where there may be multiple instances of the same pattern of components.

The state effect relationships between state variables is expressed at this time through the use of the `SV::affects` method. Call this method on the affecting SV passing as an argument the pointer to the affected SV. The `SV::isAffectedBy` method provides a way to query the state affects graph topology.

The MDS framework provides executable components for the planning and execution functions in the `Mds::Fw::Sa::Run` package. As part of your system instantiation you will need to create instances of these components and schedule them for execution.

Table 6: Planning and Execution Components

<i>name</i>	<i>requirements</i>
<code>Mds::Fw::Sa::Run::Executive</code>	Responsible for temporal constraint network propagation and time point firing. Run period should be determined by the tradeoff between required time point firing precision and available resources.
<code>Mds::Fw::Sa::Run::XGoalChecker</code>	Responsible for monitoring the status of Goals and responding to Goal failure. Its

<i>name</i>	<i>requirements</i>
	period defines the minimum response time to a goal failure.
Mds::Fw::Sa::Run::ElaborationManager	This should normally run at a lower rate and priority compared to all other components.

5 Runtime Scheduling and Execution

There are two distinct kinds of execution and scheduling that occur in an MDS runtime system. The first is the goal network scheduling and execution that is handled by the MDS state architecture and frameworks. The second is a lower-level CPU scheduler that is responsible for doling out CPU cycles to various executable components. The MDS architecture leverages the concept of a *component software architecture*[12] where formal compositional rules are used to define where different threads of execution can and cannot go. The component architecture frameworks used in this release of MDS are very lightweight, and try to avoid making too many assumptions or placing too many constraints on how components actually map to threads of execution in the deployed system. The framework provides an interface (base class) for periodically-executed components in the Mds::Fw::Run package as shown in Illustration 3. RateGroup components can be scheduled to run on separate threads (one thread per group).

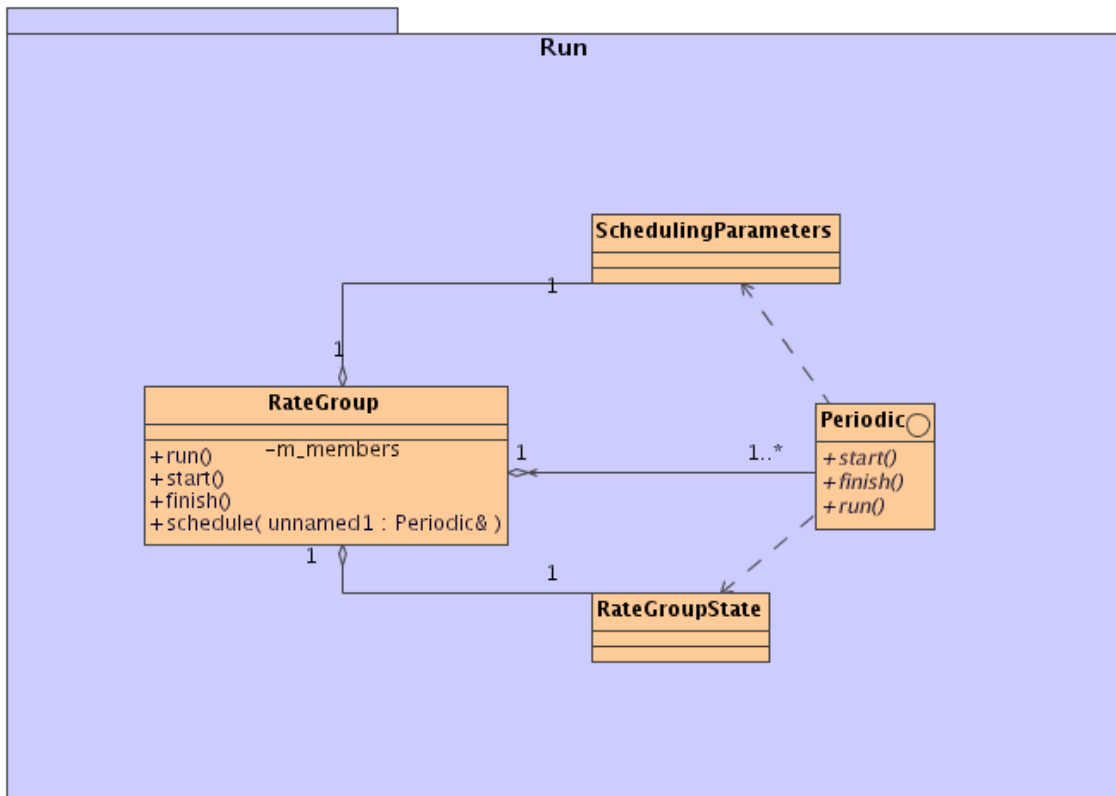


Illustration 3.: Rategroup and Periodic Component Interface

Achievers (Controllers and Estimators), and Hardware Adapters that need to execute periodically should implement the Periodic interface, and be scheduled as a member of a RateGroup. State Variables are always passive components: that is, everything that is done in a state variable is done via a function call on the thread of another component. Not all Achievers will need periodic execution, but periodic execution is a typical case in control systems. Achievers that are implemented as periodic components need to do a limited amount of work on each run() call, and return. The amount of work they can do is limited by the complexity of the system, the amount of processing power available, and the realtime timing requirements on the control loop the component is a member of.

The MDS framework provides a couple of simple rate group schedulers in the Mds::Fw::Run package. The simplest of these is a single-threaded virtual time scheduler used in most of the example tests. The general design of the schedulers in this package includes three types of components: a rate group container, a scheduler, and a time driver.

The rate group container is a simple container that works with any of the schedulers. When its run method is called it calls run on each of its registered Periodic components in the order in which they were registered. The rate group container is also responsible for subdividing the call period in order that different rate groups run at different rates. At construction time each rate group component takes an argument that defines its period. The default value is 1. It subsequently will perform the iteration of run over its members the first out of every period number of calls to its own run method.

The scheduler is a container for rate groups. Its job is to call each rate group once for every time its clock input is called. A realtime scheduler would use a physical clock of some kind as the timing source, while (as shown in the examples using SimpleTicker) a virtual time scheduler would just trigger the clock input as fast as possible.

Mds::Fw::Run::Scheduler is an example of a more complex realtime scheduler that provides a separate execution thread for each rate group, and uses an alarm signal from the operating system (via SignalTicker) for the timing source.

Your particular system may require more advanced runtime schedulers than the simple one provided in the framework.

6 Adding Elf Events

The MDS Event Logging Facility (ELF) provides a framework service for efficiently reporting generic event messages via telemetry. It includes a front-end interface for use in embedded system code, and a back-end message handling facility that can be configured to direct and manage collected event data. It also provides a flow-control mechanism that enables the suppression or filtering of events by severity or source.

Elf defines an *event message* class as a C++ class that encodes a message payload (content type). Provides one or more constructors that accept content data in its original form, serialization methods to write this data to a byte stream, deserialization constructor to read it back, and print methods to convert the data to human-readable messages. An event message class derives from a framework base class (Event). An event class defines a particular format or structure of a message and can be used to describe different kinds of events as long as they have the same physical structure.

An *event generator* is defined as a class specifically created for the purpose of reporting a particular kind of event from a particular software package. It: conditionally constructs

event messages and sends them to the event handling service for routing. Each generator can be individually configured through the event logging service to suppress or filter message generation. A generator class derives from the framework base class `Generator`, which provides the hooks to the Elf event message collection and routing services.

For each event you want to report you should first define a new event class (or reuse an existing one) that can efficiently store the given message content. Note that an event class includes a separate method for converting the event instance into human-readable (string) form. This makes it possible to create event classes with enumerated messages, where only the enumerated value (int) is actually stored in the event object, but that can automatically be converted into a human-readable form. Each new event class requires a registered instance ID in the `Mds::Fw::Sid` database.

Next, you need to define a generator class whose instance will provide the interface between your code and Elf for reporting events. A generator instance must be created somewhere in your code in order for it to be used to create events. Usually, it is convenient to create these instances within components where they will be accessible only within the component, or in a subsystem container, where they would be accessible from multiple components.

Each generator instance is required to have an instance ID assigned in the `Mds::Fw::Sid` database to keep it distinct at runtime from all other generators interacting with the Elf back-end.

Elf provides thread synchronization mechanisms between the generator and the Elf back-end. This design assumes that each generator instance should only be accessible to a single thread. If you create a generator that could be accessed by multiple threads then you may need to add your own synchronization mechanism at the generator interface (though this is not recommended).

The default back-end event handler is a simple in-memory container that is intended for capturing events that might be reported prior to the formal initialization of an Elf back-end. The MDS 2005 delivery package also includes an event handler intended for use in test systems which will format event messages and write them to a file. This is what is used in all of the example adaptations.

In a real embedded system you should plan to write a separate data handling component to replace this test adapter.

7 Initialization

Polyserializable classes need to be initialized. This includes Constraint, Measurement, Command, StateValue, StateFunction, and Elaborator. Also, every class that derives from these base classes needs to be registered with the base class deserializer before any of the serialization methods will work properly.

8 User Interface

Some primitive user interface support is provided in MDS release 6.1. The goal network provides a utility method `Network::saveXML` which will produce an XML-formatted description of the goal network. This method can be called at any time on any goal network (proposed, scheduled, or executing), and the output will be written to an external file. The XML file can then be used for analysis or to generate graphical depictions of the goal network. An example graphical translator is provided in the test helper `MDS::GoalNet.pm`, which can be found in the `verification/Helpers/MDS` directory in the source tree. An example test that uses it is described in Appendix A.

9 Data Management and Transport

The MDS planning and execution frameworks don't make any strong assumptions about how data will be stored persistently or transported into and out of the control system. However, frameworks are provided to support storing and transporting data in the form of files or data products (abstract container classes that can be stored as files) containing serialized data objects such as Measurements, Commands, Goals, or State Functions. The intent of the design is that value history containers would provide an interface that enables a data controller to move data between the in-memory container (the value history container itself) and more efficient and persistent deep storage by storing them in data products on some external media. This external data product format is also ideal for use as a transport artifact. The serialization process can include various forms of data compression.

The value history to data management interface is not implemented in the MDS 6.1 release, and the current examples do not include any examples of data states. A more detailed description of the design can be found in [11].

10 Porting to Physical Hardware

The software development process described here encourages you to develop the control system logic in a workstation environment and verify it against simulations before attempting to test it in-situ in an embedded system with physical hardware. This process helps to ensure that many complex software coordination details that would be difficult to diagnose in a system context have already been verified before they are brought into the context of physical devices.

The interfaces between an MDS control system and physical hardware are primarily contained in the hardware adapters or underlying device drivers. As described earlier, you may wish to develop separate hardware adapters or device drivers to support simulated devices and physical devices. When you do this, be sure to use a common interface, and to develop interface tests that ensure that the two implementations are interchangeable. It is also highly recommended that you develop thorough physical and simulated device tests that can verify that the physical device actually works according to its requirements and models, and that the simulation works the same as the physical device. These two areas represent a common source of errors in embedded control systems that can be exceedingly difficult to diagnose in a full-up system context. Developing and verifying these low-level device interface tests will also require you to configure the build system to support cross-compilation to the target platform, and to develop or port any supporting services needed to execute the test system on the embedded target. Most embedded operating systems support remote development and testing interfaces that would allow you to load and execute your software on the target via a network interface. The MDS test harness in this version provides vestiges of mechanisms that were used to support remote testing on VxWorks and realtime Linux embedded targets.

11 Optimization Notes

MDS philosophy is to first get it right functionally and then optimize from there. Recognizing, however, that there will be strict limits on memory and processing resources in an embedded control system, the state analysis process provides opportunities for the system engineers to use simplified models, and less precise representations as needed to make the system practical.

Implementers should try to review requirements in order to have implementation considerations reflected in the state analysis process as early as possible so that when changes must be made, time remains to consider all of the effects of the change on the rest of the system.

Implementers are free to consolidate logic within required methods as long as the specified outcomes are achieved. For example, the state analysis specifications for goal merge and subset rules will usually appear in the form of a cross-product table of inputs and expected outcomes rather than as an algorithm. The implementer is free to find an efficient algorithm to achieve the given outcomes.

11.1 Memory Management

Dynamic memory allocation, while a great convenience in the C and C++ programming languages, can lead to problems if not used carefully in programs that have realtime requirements, or use multiple threads. Many default heap managers are not thread safe, and at best are unlikely to provide fast and deterministic timing when performing arbitrary allocations. Furthermore, over time, a shared heap can become fragmented, leading to degrading performance over time, and eventually a system fault.

A common strategy to address these problems is to divide and conquer. That is, to preallocate pools of memory for use by separate components or threads so that thread interactions in a common heap are avoided. In most cases the pool management can be further simplified by keeping the size small and the pools used to allocate objects of fixed size.

In the MDS software architecture value histories are defined largely to take responsibility for these sorts of memory management issues. State variables act as synchronization points between components that may be executing on different threads, and the value histories within the state variables are intended to provide the required synchronization capabilities. The architecture also stipulates that state variables (via their value histories) are the only place where state information is allowed to reside.

The examples provided in the sample adaptation (appendix A) demonstrate the use of small memory pools of one or two latest values. A larger (deeper) value history might use a circular buffer, or least recently used algorithm to provide deterministic allocation.

11.2 Optimization Strategies and Considerations

- value class representations: float vs double, runtime value vs serialized value, cost of conversions needed for algorithmic use, depth of history, compressability and telemetry requirements
- algorithmic costs (control and estimation): generally independent of any MDS framework
- pay most attention to logic in components that run at highest rates, or have shortest deadlines; secondly to components that process large amounts of data (deep histories, frequent i/o, large value types).
- Use memory pools for value histories. The frameworks provide a couple of example of pooled-memory value history containers in the `Mds::Fw::Sa::Vh` package. These containers are also async-safe through the use of atomic methods.
- Passing reference-counted pointers versus copy-by-value in value history queries: Value history query interfaces are used frequently in the MDS architecture, and this makes them an important target for performance tuning. Performance benchmarks have shown that for small value classes containing no more than a few words of data, it may be faster to copy results by value than it is to return a smart pointer. For states that have large value representations, or that have polymorphic values, the return by pointer may be the only option, and this is why the default SV interface requires a return by reference-counted pointer interface. For states with small values, however, the adapted state variable may implement additional query methods for use by its achievers that can perform return by value.
- Be careful of thread synchronization overhead.

12 Testing

Frequent and early verification is essential to an efficient development process. The development procedures described in earlier sections include steps for unit testing and system testing the adaptation. This section describes in more detail how to write those tests.

12.1 Writing Tests

Although you are free to develop tests that work outside of the MDS test harness, you should be aware that the MDS test harness was developed to support fully-automated distributed multi-platform system tests. The test frameworks included in the distribution may not support your particular system test configuration, but it is usually worth the effort to automate tests so that they can be run more often and produce more objective results. Section 9.2 provides more details about how the test harness and included test frameworks work to help you design and implement a system test configuration.

12.1.1 Unit Tests and System Tests

A unit test is defined as a test that is intended to test a particular piece of the system in isolation. Usually, these tests are simple programs that exercise the API for a class or subsystem, though they can be more complicated than that. The test harness considers a unit test to be any test where it simply has to execute (and optionally build) one program and then account for the results. System tests can involve launching and coordinating the execution of multiple programs, potentially on multiple hosts. For example, remote deployments can be launched via ssh or rsh, or an embedded system such as VxWorks embedded host launcher¹.

The test harness also supports build tests whose purpose is to construct the libraries or executable targets that unit tests or system tests will invoke. Note that these types of tests are merely ways of describing the purposes of various kinds of tests. To the test harness, they are all just procedures for it to run, and in fact it is common practice to have system and unit tests also build some of the artifacts that they need to execute, and express dependencies on tests that build other common frameworks. The delivered system also includes “tests” that perform various tasks such as extracting API documentation using Doxygen, creating archive products, and installing external libraries.

All tests that run under the MDS test harness are expressed as a script in the verification subdirectory. Test scripts are usually organized in subdirectories under the verification directory that parallel target package directories in the source tree. For example, tests that apply to code in the Mds/Fw/Time package will show up in verification/Mds/Fw/Time. These scripts can be written in perl or any shell language (sh,

¹ The examples provided in MDSV6.1 don't exercise any of these system test frameworks, and some of them may be obsolete.

ksh, bash, csh), though perl is the language in which the test harness and most of its helper modules is written, so perl should be considered a preferred test script language.

Test scripts invoke one or more executable programs from the delivery directory, which are the result of invoking build rules for source code in the source directory. With the help of test helper modules, test scripts can use the build rules to build the targets they need to execute according to the particular test configuration in which the test harness is currently running.

12.2 MDS Test Harness and Test Frameworks

Tests are invoked by calling the test harness script (verification/TestMaster/runtests.sh) with the test scripts you want to run passed as arguments on the command line. The test harness will determine dependencies between the given set of test scripts using any dependency files it finds in the same directories as the scripts. The dependencies may cause other dependent tests to be added to the list of tests to run, or determine a particular order in which they must be run. If there are no dependencies, and the local system configuration allows, some tests may be executed concurrently (see TestMaster reference manual for details). What the test harness actually does is create a makefile expressing each test script as a make rule, and then runs make.

The test harness will create a top-level output directory for all of the results. The name of this directory can be specified through the -i and -b command line options. Then, for each test script it runs it will create a subdirectory under that output directory in which it will run the test and record any outputs. The test harness automatically redirects the standard output and error streams from any executable program it runs into files in the output directory (executable_name.stdout and executable_name.stderr).

The test harness also controls the environment in which the test scripts and executables are run. The test harness filters out most environment variables defined in the user's environment and defines environment variables that describe the test configuration specified in one or more -g command line options, or as assignments on the command line. It is usually most convenient for you to create one or more top-level configuration files that define your particular test configurations. These top-level configuration files can include settings from other files, including system configuration settings in the verification/TestMaster/Config directory.

It may be convenient to use a top-level script or makefile in the MDS_ROOT directory to invoke a set of tests that are used routinely during development and testing. Using a makefile in this way is particularly convenient, because then you can invoke common tests using a short simple alias name. The delivery system includes a makefile in the MDS_ROOT directory that does this for the included system build and regression tests. You can add your tests to this makefile, or create your own.

12.2.1 Test Code Organization

It is important to keep test code distinct from code that you intend to be part of the actual delivered system. The convention used in the MDS source tree is to put most test code in a separate Test subdirectory under the subdirectory of the target package that it tests. Code in a Test subdirectory is defined in a Test sub-package under the target package. This makes it easy to tell that everything in this subdirectory is test code. This way it is also easier to create a TARGET_TEST_LIB that provides some common test frameworks for a number of unit test executables in the test package.

12.2.2 CppUnit Unit Test Framework

CppUnit is a C++ unit testing framework designed initially as a port of the JUnit java unit testing framework. More information about CppUnit can be found at the CppUnit homepage at <http://cppunit.sourceforge.net/cppunit-wiki>. You don't have to use CppUnit to write tests, but you may find it useful. Primarily it provides a standard internal interface to define a test, and a simple process for running each test and accounting for the results. An example CppUnit unit test program is provided in source/Mds/Ra/Example/Test/Standalone.cpp. This example provides comments to describe what each part does. The example explains the basic interfaces to CppUnit

12.2.3 Test Scripts and Script Helpers

The simplest kind of test script is one that simply invokes one or more unit test executables. Since this is a very common pattern, it is the most thoroughly supported by the test harness. You still need to write a test script for this, but you can write one test script that can build and run all of the unit tests defined in a given source directory. An example script that can do this is provided in verification/Mds/Ra/Example/RunTest.pl.

This script can be broken down into four key lines. First, the top line is a shell script redirect that declares this to be a perl script:

```
#!/usr/bin/perl
```

After several lines of comments there are two lines that do most of the work. The first tells perl where to find helper modules. The second one loads the MDS unit test helper. This is a perl module containing a number of methods for invoking the build system, and reporting results back to the test harness.

```
use lib split(/:/,$ENV{TSTHELPERPATH});  
use MDS::UTest;
```

The last line is the actual specification of the test. This says to build and run a particular unit test executable and report its results. It calls the `MDS::Make::build` function to build the unit test executable target (identified by naming convention as “`Mds/Ra/Example/Test/Standalone.cpp`”), and then execute the result as a unit test. This test assumes that there is a `make.cfg` file in the `source/Mds/Ra/Example/Test` directory that contains a `UTESTSRCS` directive that lists `Standalone.cpp` as one of its unit test sources. The test also depends on you having already run the `metamake` test to find all of these targets in the source tree.

```
MDS::UTest::runTestTarget(  
  MDS::Make::build(  
    “Mds/Ra/Example/Test/Standalone.app”) );
```

Alternatively, you could just let the test helper find all the unit tests itself and run them all:

```
MDS::UTest::buildAndRunTests(  
    "Mds/Ra/Example/Test");
```

Here, the test helper will try to build and execute a unit test executable for each source file listed in the UTESTSRCS in the named subdirectory's make.cfg file.

Another useful test helper is the MDS::Elf module, which can be used to analyze events reported through the MDS Event Logging Facility (Elf) framework. Simple unit tests will ordinarily route events to an output file called *Elf.log*. This test helper provides methods to analyze the log file particularly for goal planning and execution events, and then assert requirements such as time point firing order and timing. [examples...]

12.2.4 Testing MDS Adaptations

Testing different levels of integration tends to require different amounts of test support. At the simplest level of individual classes and methods, very little test framework is needed, while testing full control loops requires quite a bit of additional test support. This section describes patterns for three categories of tests along with the MDS frameworks and test-support features provided to help implement them.

12.2.4.1 Class API Unit Tests

True unit tests are supposed to test individual units of functionality at an atomic level before those atoms are integrated into larger subsystems. This sort of test is recommended for all of the data classes that are used in MDS adaptations including Measurement, Command, StateValue, StateFunction, and Constraint, particularly for methods in Constraint and StateValue used to merge and compare values. These methods can be subtle and complex. Errors in these methods can be very difficult to diagnose in a running control system, while they can be very easy to test at an API level.

Unit tests should exercise all of the constructor signatures of the target classes you intend to use in the real system, along with any other methods you will depend on. In particular, the merge methods and subset methods in Constraints are very important to test in all the combinations. Since these tests require only the target classes, they can be developed

concurrent with the development of the target classes, adding tests for each new method as it is implemented.

12.2.4.2 Planning and Scheduling Tests

Planning and scheduling tests are used to exercise and verify goal elaboration functions, the implementation of planning and scheduling methods, and as a prerequisite to execution tests. Tests should be devised first to verify the schedulability of nominal operational goals, and then to verify that unallowable goals or goal orderings are rejected. Note that because many of the default implementations of planning methods in the framework base classes for SV and Achiever return true, it is important to implement additional tests to verify that scheduling fails for goals with unachievable physical or temporal constraints, or where transition goals are missing. These tests can be implemented as the planning and execution methods are implemented in the adapted code.

Scheduling tests require that the system instantiate all of the planning and execution framework components (Executive, XGoalChecker, ElaborationManager) as described in section 4. Top-level Goals to be scheduled can be passed directly to to ElaborationManager's internal elaborate method, which provides a direct result (the method that takes a single Goal as argument is the external method normally used in operations which will enqueue the Goal and elaborate it later by calling the internal method).

12.2.4.3 Execution Tests

Once the basic planning methods are implemented and tested you can proceed to test execution by promoting the scheduled network. This can be done in your test by directly promoting the network you created using the method described in the previous section and then letting the system run by advancing the component scheduler's clock.

Alternately, you can do this using operational interfaces which provide more indirect feedback. The operational interface requires that the top-level Goal be enqueued in the ElaborationManager's input queue using the external elaborate method (the one that takes just a Goal argument), and then running the system. In this case, results will be available as event messages in the Elf event log output file. The system should produce numerous messages as the goal is elaborated and scheduled, and then promoted.

Once the goal network is promoted messages will be generated when a time point's execution window opens, and then when it fires. Section 6 describes how you can add your own Elf events to your code to report significant events which can also aid verification. Section 10.2.3 describes some test helpers that can be used to analyze event messages in the Elf output.

12.3 Debugging

Ideally, the process described here for developing code against a structured sequence of test cases starting with low-level unit tests, and proceeding up through complex system tests, should simplify the process of diagnosing problems at each step. Liberal use of assertions in the code will also help you discover problems at the earliest possible opportunity, and the earlier a problem is found, the easier it is to diagnose. Properly written tests (particularly unit tests) will clearly indicate the particular failure, and an understanding of what each test case is verifying should point you directly to the problem (so, it is important to document the test cases sufficiently to make the requirements clear).

When that fails, you have to dig more deeply to sort out what happened. The MDS test harness provides some help by saving all of the test outputs in separate, organized directory structures for each test. Also, when a test fails, the test harness writes a debug script in the output directory (called debugMe) that can be used to re-launch the failed test application under the debugger. Normally, all you have to do is run the script to launch the debugger, and then, using debugger commands, run the application to repeat the failure.

12.4 Simulation

The MDS version 6 release contains no simulation frameworks. Simulations are important for developing control system software either before the hardware system is finished, or when the actual embedded system is just too precious a resource to use for testing software. The recommended approach is to develop separate hardware adapter components or underlying device drivers to act as interfaces to simulated hardware. Usually, hardware adapters are just thin interfaces to underlying hardware or device drivers anyway, so having alternate versions to interface with real or virtual hardware requires little extra effort. The MDS test harness can support alternate switchable system

configurations like this through the use of test harness environment variables, and conditional compilation.

13 Advanced Topics

13.1 Multi-Threading

Embedded control systems frequently rely on Multi-Threading to achieve concurrent execution of multiple tasks. The MDS frameworks do not assume any particular threading model or implementation (though the example threaded rate group scheduler does use posix threads), and are intended to be usable under different execution frameworks. The current system makes some design-level assumptions that the execution framework will be based on a periodic execution model rather than a purely event-driven one. The components could probably be adapted for use in an event-driven system, but this is not provided in this release.

The framework design assumes that State Variables, and specifically their Value History containers, serve as the synchronization points between threads of execution. The example value history containers provided in the framework provide async-safe update and query interfaces when used as described in their class documentation.

Furthermore, the planning and execution components provide inter-component interfaces that are intended to be thread-safe when executed according to their documentation.

Appendix A. Open-Loop Temperature Controller Adaptation Example

The MDS V6.1 distribution provides an example adaptation that includes a set of requirements developed through the state analysis process, and the corresponding software that implements and verifies those requirements. The example demonstrates a simple open-loop temperature control system that uses the temperature of a system indirectly by controlling the states of two heater switches. This example was chosen to elucidate the analysis and development process, and to demonstrate some of the integrated fault protection mechanisms in the MDS architecture. It is not intended to be particularly realistic as a physical system (modern devices can easily provide closed-loop thermostatic control in hardware), or to provide a complete fault analysis and recovery of the given system. And, the software is not as highly optimized as it might need to be in a fully-integrated embedded system. Rather, the code is written to be instructive, without being overly elaborate.

Getting Started

The source code for the MDS V6.1 frameworks, including the temperature controller example are included on the distribution disk in the source.tar archive file. Instructions describing how to extract, install, and build this code can be found in the build.html file in the root directory of the CD. Follow the instructions provided there to build the framework libraries and run regression tests before proceeding any further.

Once you've got the system installed, locate the example source code in the source/Mds/Ra/SimpleThermostat directory.

Before proceeding to look at the source code, let's first examine the requirements which include a description of the system being controlled, and the control problem being solved. The requirements are documented in two files provided on the distribution disk. These specifications represent standard artifacts of the state analysis process. The main requirements document can be found in doc/example/tcanalysis.pdf. It describes the

physical system under control, all state models, estimation and control algorithms, value and constraint representations, and required verification tests. Details of the required planning and execution methods are specified in a spreadsheet which can be found at `doc/example/methods.xls` on the distribution disk. You'll need a Microsoft excel-compatible spreadsheet program to view this file.

First, it is important to understand the physical system that this adaptation is trying to control. The requirements document describes this in the first section after the introduction. The system consists of a hardware device with some thermal mass containing two heaters which can be controlled through software-commandable switches. There is a temperature sensor to measure the temperature of the device. The device resides in a cold environment so that at least one of the heaters needs to be on in order for it to achieve a warm temperature. This might represent an instrument that needs to be warmed up to a calibrated operating temperature.

The control system for this device requires state variables to represent the two switch states, the temperature state, and the sensor health state. In this example, temperature control is achieved using an open-loop control algorithm: goals on the temperature state have to be elaborated into goals on the directly-controllable switch states. To achieve closed-loop thermostatic control the system would have to be able to delegate control of the switches to a controller associated with the temperature state. In this case, delegation was not used².

A state effects diagram is shown in illustration 1. This shows that the heater switch states affect temperature, and the sensor health state affects the quality of temperature measurements. The physical temperature state also affects the temperature measurements.

The subsequent sections in the state analysis document describe the physical models of the switches and sensor, and algorithms for representing and estimating the various states. Review these descriptions so that you will be able to follow the logic when we start to review the code later.

One of the last items in this document is a collaboration diagram that describes the required control loop element components and their interactions. Note that the architecture includes a simulation component to take the place of the physical hardware at the level of a device driver (below the hardware adapters). Hardware adapters for the

² Delegation interfaces are not implemented in MDS V6.1 release.

two switches are combined into a single component. This is because the hardware interface to the two switches is likely to be a single physical device attached to the computer and accessed through a common device driver. In such situations, it is often convenient and efficient to design hardware adapters that respect the physical organization of the hardware interfaces. Note that for a similar reason, a single switch state estimator is used to estimate both switch states. The switch state variables remain separate because we want to be able to elaborate and schedule goals on each switch state separately.

Finally, at the end of the state analysis is a list of test cases that will be used to verify the basic functionality. This is intended to be only a representative set of functional tests. There are many more sub-cases that could be tested, and should be verified in a real adaptation. Also, in a real system you would want to extend the verification process beyond the simulation stage, and verify the software in-situ in the embedded system.

The second part of the requirements specification is contained in the spreadsheet. As described in the state analysis course, the spreadsheet is a convenient form for analyzing the requirements for methods that govern the interactions between two state constraints, or a state constraint and state value. We will refer back to these tables in subsequent sections when we look at individual classes that implement these requirements.

A Tour of the Software Adaptation

Find the example software adaptation of this control system in your installed source tree in the directory `source/Mds/Ra/SimpleThermostat`. Note that the directory is organized into a main body of code, plus three sub-packages, each in separate subdirectories. The main body of code in the `SimpleThermostat` directory contains the adapted components and support classes. The `Goals` subdirectory (sub-package) contains the adapted goals, elaborators, and tactics that apply to these states. It is convenient to put these in a separate package because the goals and elaborations are likely to be adapted further as testing, and particularly, operational testing begins, while the underlying control code should remain stable.

A `System` subdirectory contains a primitive system architecture class that is responsible for instantiating and initializing the control architecture in this simple example. In a more complex system you might use hierarchical aggregation of subsystems following

this pattern, rather than doing it all in a single class. In this case, the subsystem architecture assumes that it will be instantiated in a main function provided by the system test architecture.

A third subdirectory contains the classes that support the unit and system tests. These are in a separate package intended to keep the test code distinct from the target code. Note that these classes are all heavily commented to help explain various implementation details.

Let's look at the main body of code first.

First of all, you should be able to locate all of the required control loop components as separate classes:

State Variables:

TemperatureSV	Temperature State Variable
HeaterSwitchSV	Common class for heater switch state variables
SensorHealthSV	Temperature sensor health State Variable

Hardware Adapters

HeaterSwitchHWA	Switch hardware adapter
TemperatureSensorHwa	Sensor hardware adapter

Achievers

HeaterSwitchController	Switch state controller
TemperatureEstimator	Temperature state estimator
SensorHealthEstimator	Temperature sensor health esimator

Reporting/Telemetry

Event	Telemetry event message
EventGen	Interface for reporting telemetry events

Simulation

SimComponent	Simulates hardware devices and physics
--------------	--

The remaining classes are support classes that represent the data types and containers used by these components:

TemperatureMsmt	Temp sensor measurement class
TemperatureStateVal	State value for temperature state
TemperatureStateFunction	State function for temperature state
TemperatureConstraint	Temperature state constraint
HeaterCmd	Heater switch command
HeaterSwitchStateVal	State value for switch state
HeaterSwitchStateFunction	State function for switch state
HeaterSwitchConstraint	State constraint for heater switch
SensorHealthStateVal	State value for sensor health
SensorHealthStateFunction	State function for sensor health
IdealTempModel	Temperature model functor

Note that the physical model for temperature as a function of known states and time is explicitly represented in a separate functor class (IdealTempModel) because it is used in more than one place, and to make it explicitly distinct when it is used in the switch state estimator.

The Goals directory contains classes to support two macro goals for this subsystem.

KnowAll	Elaborates into knowledge goals on all four state variables over a given duration. This goal is used to establish background knowledge required to plan and schedule control goals on these states.
StayWarm	Implements the required goal to maintain a warm temperature state over a given duration (between two time points). The goal has two tactics, as required, to use one or the other heater to achieve the required temperature. Note that the associated Tactic classes are implemented in the same files as the Elaborator.

Examining the Adapted Classes in More Detail

Let's start by working from the subsystem level down into the more detailed implementation classes. Along the way we'll take a look back up at how we instantiate the subsystem in a test harness environment in order to verify the functionality. The temperature controller subsystem is constructed in the ThermostatArch class in the directory `source/Mds/Ra/SimpleThermostat/System`. The declaration in `ThermostatArch.h` declares public member instances of each of the state variables. The associated estimators, controllers, and hardware adapters are declared as private members. In this example the simulation component is also declared here. In a more complete example, references to the simulation would probably be compiled conditionally (inside an `#ifdef` directive), and there might be alternative physical deployment device drivers instead. Because the simulation is included, this class also provides a few test ports in the form of public methods that can be used to induce various failure modes in the simulation. Use of these methods will be explained later when we look at the test code.

All of the logic in this class is in the constructor. This is the method in which the architecture is actually instantiated and the inter-component connections are established. These connections are all specified as requirements in a collaboration diagram at the end of the state analysis report. Most connections are represented as member pointer or reference attributes in each component class, and have to be explicitly initialized in the separate component constructor calls. For example, the heater switch controller needs to have a connection to the switch hardware adapter, and to each switch state variable. These connections are initialized in the initializer list in the implementation of the architecture constructor.

The connections from state variables to their associated achievers are supported in the framework base class for state variables (SV). The calls to `SV::linkAchiever` on each of the state variable instances in the architecture constructor body initializes these connections.

The second thing the architecture constructor does is to schedule each runnable component in a rate group. In this example all of the components are scheduled to run in one common rate group which is passed in as a constructor argument (this design presumes that a higher-level subsystem will create the rate groups and schedule them with a scheduler). The components are scheduled to run in the rate group in the order in which they are passed as arguments into the `group.schedule()` call.

Finally, the last thing the architecture constructor does is to define which states affect which others by calling the `SV::affects` method on affected state variables. For effects that cross between subsystems you may need to make these calls at a higher subsystem level. The graph of state effects that is established using these calls is important to the goal scheduling algorithms used in the frameworks.

Next, let's look at some of the classes where these components are implemented.

State Variables

The requirements specify that some of the planning and execution methods are to be implemented in the state variables, and others in the associated achievers. In most cases, the state variable implements the projection methods, and uses the default implementation of other methods to call through to implementations on the achievers. In other cases such as the temperature state variable, many of the methods are overridden and implemented in the state variable itself because there's only one achiever and the methods don't have any particular dependencies on the estimator class.

Note that the state variable's value histories are not constructed or initialized internally. This is a design choice. In our example, all history initialization is performed by the associated estimator when it is initialized at runtime through the scheduler start method. For states that can be initialized statically it makes sense to do so at the earliest opportunity. Some states may require access to underlying data management frameworks in order to retrieve persistently stored checkpoint data. In that case it would be better to wait until a point in the initialization process when all of these lower-level services have been started (whether or not that has happened by the time you construct your

architecture is a design choice). Still other states may require an active estimation process to establish valid estimates. In such cases, the initial state should be unknown.

In the example adaptation we initialize most states to unknown. In the first version of the analysis we realized that it would be impossible to schedule any of our control goals that require knowledge of switch and sensor health states if we didn't have current estimates of those states. To solve this we schedule a background activity that starts in the setup part of each test. The background activity schedules knowledge goals on each state variable will be running when we try to schedule and execute the target control activities.

Estimators and Controllers

Our example provides only one instance of a controller implemented in `HeaterSwitchController`. The controller runs periodically (at the same rate as the other components) and issues commands to try to bring the switch states into compliance with its given state constraints (via the ordinary constraint in a given `XGoal`).

The `HeaterSwitchEstimator` provides an example of a reasonably complex health estimator that has to try to infer the health of the switches from indirect evidence in the form of most recent commands, and their effects in the temperature state history.

All of the estimators and controllers are implemented so that when unconstrained they will return immediately from their periodic run calls and effectively do no work. A minimal knowledge constraint is required for the estimator to begin estimating and updating its state variable, and an achievable control goal is required before the controller will do anything.

Hardware Adapters

The hardware adapters in our example are defined as passive interfaces to underlying hardware (probably via a device driver that isn't explained in the model). The interface is passive in the sense that all interactions with hardware will occur on the public command and measurement calls. The only thing the hardware adapter has to do here is to remember a history of recent commands and measurements needed to support the required estimation algorithms. The requirements specify that we only need to remember one most recent measurement (in fact, even that could probably be optimized away), and the one most recent command for each switch.

Since the hardware adapter's value histories are intended to also serve as interfaces to telemetry, they should not be entirely optimized away unless it is certain that the given values will never be needed for remote diagnosis, or can be obtained by other means. Note that this telemetry interface is not implemented in the value history interface in MDS V6.1.

Data Classes: Measurements, Commands, and State Values

Data value classes represent the kinds of information passed between components in our example control loop. `TemperatureMsmt` represents the one measurement provided by the hardware adapter. This class contains a one-byte raw data value and a boolean flag to reflect the quality of the measurement. The raw data value is specified precisely in the requirements. The `m_isValidMsmt` flag is a placeholder value that, given the specific requirements, could be optimized away. The MDS control architecture specifies that measurement interfaces should always provide an indication of the validity or quality of the measurement. This is to ensure that the user doesn't have to guess or infer whether or not the hardware is even turned on. In this case the requirements stipulate that the measurement itself will provide an explicit indication when the sample is invalid, so the extra flag is superfluous. In most cases, though, the raw value sampled from hardware isn't likely to provide an explicit indication of validity, and the extra flag would be needed in those cases.

The Measurement base class provides a time tag for every measurement instance.

Most of the methods implemented in the `TemperatureMsmt` class are simple and straightforward. The only methods that might require some explaining are the serialization methods, `writeObject`, and the deserializing constructor (the one that takes an `ObjectInputStream` argument). These methods implement a common virtual interface on all data classes that are intended to support portable storage and transport of these values in telemetry. Although the telemetry interface is not implemented in this release, the design relies on using symmetrical input and output interfaces in each data class to provide conversions to and from a bytestream representation that could be stored in a file, or transported as a data stream over some communications medium. The `Mds::Fw::Ser` data stream frameworks provide support methods for portably reading and writing standard primitive data types. This method eliminates the need for external tables or mappings between flight and ground implementations of data structures.

HeaterSwitchCmd represents the one kind of command used in this control system. Since we use one hardware adapter and one command type to command the positions of two switches, we chose to include a switch identifier attribute in the command value. This way we can maintain a single command history in the hardware adapter and still know which command went to which switch.

State values are implemented in HeaterSwitchStateVal, SensorHealthStateVal, and TemperatureStateVal. These classes represent the pure state values of their separate physical states at a point in time. The implementations of these classes is similar to those of the measurement and command classes except that state values also need to provide an indication of uncertainty. Since they represent the result of an estimation process these values are never absolutely certain. So, the architecture requires that values include an explicit indication of uncertainty. For scalar values such as the temperature state the uncertainty is represented in two distinct ways. In our temperature state value, the class stores a mean value, but provides an interface that expresses the value as a uniform distribution between a max and min value. In the example, the deviation is defined as a constant value that is a function of the underlying device. The value class also provides an isUnknown method that return true if the uncertainty is greater than the ordinary measurement uncertainty. For example, this provides a default state value when the estimator has not yet estimated the state.

This particular adaptation chooses to maintain a piecewise-continuous state value history in each state by using a framework-provided ConstantStateFunction template class as the state variable's value history type.

State Constraints

State constraints are like other value classes in that they have to implement serialization methods. In addition, they need to implement two key methods that are used during the planning and scheduling process to determine if goals are mergeable and achievable. Requirements for the outcomes of the mergeFrom and isSubsetOf methods are specified in the spreadsheet that accompanies the requirements document. The mergeFrom method modifies the object constraint to turn it into a constraint representing the intersection of its original value and the value of the argument constraint. That is, the resulting state constraint describes a set of target state values that is the intersection of the set of states permitted by the original constraint and the argument constraint.

Requirements for these methods are usually given in the form of a table where the rows and columns describe the original object constraints's values, and those of the argument constraint. The cell values describe the result of the combination of the row and column values.

The MDS architecture provides unique definitions and rules for control constraints and knowledge constraints. This is often reflected in the requirements specifications along with a note that the adapter should implement a logical combination of the knowledge and control constraints somehow. Usually, these separate kinds of constraint are implemented in a common constraint class. In our example, there is only one kind of knowledge constraint: that the state be known (when the state is completely unconstrained, the estimators stop estimating the state). Thus, the KNOWN state in our state constraints represents a knowledge constraint in the absence of any control constraint.

Goals and Elaborations

The requirements for this adaptation specify only one goal with two tactics that need to be elaborated (others are specified in the requirements, but not implemented since the ones shown are sufficient to explain how it works). The required goal is implemented in the class StayWarm in the Goals subdirectory of the adaptation. This goal uses three other goals in its elaborations: GetWarm, Sensor, and Heater. The GetWarm goal defines a temperature transition goal. Both this and StayWarm use the Sensor goal to require the sensor to remain healthy (needed to estimate temperature state), and the Heater goal to elaborate heater switch states. The first tactics in GetWarm and StayWarm each elaborate goals on the switch states to have one switch open and the other switch closed. Their alternate tactics elaborate goals to have the switches in the opposite positions. Each of the Heater switch goals elaborates a switch transition goal ahead of itself.

The adaptation includes a second goal, KnowAll, which implements a background knowledge goal on all states. This is needed because the default states of the estimators is an idle state, and all states are initialized to unknown when the system starts³. In order to be able to schedule the StayWarm control goals the system has to be in a configuration where the health states of the sensor and switches are known. That implies that the

3 A more complete example might include a persistence mechanism so that once a failure state has been determined, the state variable remembers this persistently. The framework supporting this mechanism was not implemented in MDS V6.1, though the design is part of the architecture.

system has to be already executing a set of control goals, or at least a set of knowledge goals when the new control goals are scheduled. The KnowAll macro goal provides such a set of background knowledge goals.

The tactic functors are implemented inside the same files as their elaborators for convenience. These are effectively private implementation classes.

Note that the GetWarm and StayWarm elaborators implement different failure responses in their getReElaboratableGoals method than do the Heater and Sensor elaborators. The Heater and Sensor goals have no alternate tactics, and so when they fail they want to pass the failure up to their parent goal, GetWarm or StayWarm. Since the temperature goals can recover from certain failure cases, it does have alternate tactics, and it responds to failures by adding its parent goal to the set of goals to be re-elaborated.

Unit Tests

The recommended development process suggests that you implement unit tests for all measurement, command, state value, and especially state constraint class you implement prior to implementing any of the planning and execution methods, or trying to validate any system behaviors. The unit tests will verify that simple mechanisms like constructors and accessors are working as expected in a straightforward way where mistakes are easy to find and fix. Unit tests also make it much easier to verify all of the combinations of state constraints that might be expressed in the requirements, but never (or worse, rarely) exercised in an actual system test. Mistakes in the constraint merging or subset logic can be very difficult to diagnose in a system test when complex interactions between states are possible.

The example adaptation includes unit tests for each of the adapted data classes. All of these test classes are defined as unit tests in the Test subdirectory of our adaptation directory. Let's just look at one of these in detail. TemperatureConstraint_test.cpp uses a unit test helper class (which doesn't really do anything here). The test helper class just provides a context for several test methods that will exercise various requirements on the constraint. The cpp file implements each of these test methods along with a main method that configures the CppUnit test harness to call each of the test methods.

NOTE: Greg hasn't finished the merge and subset tests here yet.

TemperatureConstraint_test.cpp is listed in the makc.cfg file in its subdirectory in the list of UTESTSRCS, that is, as a separately executable unit test. In order to run this test by itself we need a helper script that will build and run it under the test harness. This script is implemented in verification/Mds/Ra/SimpleThermostat/Constraint/TemperatureConstraint_test.pl. This perl script is very simple. The main body of the test script is shown here.

```
$testname='TemperatureConstraint_test';  
MDS::Utest::runTestTargets(MDS::Make::build::(  
  "Mds/Ra/SimpleThermostat/Test/${testname}.app"  
));
```

The argument to the MDS::Make::build method describes a make target in the system makefile that the metamake program generated when you ran the metamake test (this makefile expresses all of the file dependencies between all input and output files in the system). By convention, this makefile uses the “.app” suffix to describe executable targets. The build method will invoke make under the test harness (using the system configuration defined by the test harness arguments) to try to build the unit test executable from the source file. Once the executable is compiled and linked, the MDS::Utest::runTestTargets method will try to run it. Since this is a unit test, this method only looks at the exit status after the program runs to determine a success status to report.

We can now try running this test from the MDS_ROOT directory:

```
> cd verification/TestMaster; ./runtests.sh -g  
gcc343x86ace541.cfg -z -b $MDS_ROOT/results  
MDS_BUILD_OPTION=debug -i out  
Ra/SimpleThermostat/Constraint/TemperatureConstrain  
t_test
```

The Makefile in the root directory also provides a shortcut for this command via the make target “tempconst_ute”. So, can also run the test by just typing “make tempconst_utes” in the root directory (once you’ve configured the default test config in the Makefile).

When you run this test it will print a lot of NOTEs to the standard output stream that describe all of the environment variables visible to the test when it is run. The most important information should be at the end of the output (once it finishes):

```
...  
TESTEXITCODE AOK 0  
TESTRUNSTATUS passed 0  
TESTSUMMARY  
Mds_Ra_SimpleThermostat_Constraint_TemperatureConstra  
int_test passed 0  
All Tests Passed
```

If the test hadn't passed, the output would indicate that a failure occurred, and it would list the names of output files that would provide more detailed results. The default test configuration described in the root Makefile directs all test outputs into a "results" subdirectory under MDS_ROOT. Each test's output will go into a separate subdirectory under that named using the make of the root Makefile target with "tmp." as a prefix. This subdirectory will contain a number of output files generated by the test harness, but the diagnostic details that the test itself generated will be in a subdirectory whose name is constructed from the test target name and ".dir" as a suffix. This directory will contain a stdout and stderr file that will include and diagnostics from the compiler and linker, and any results generated from the test helper perl script. The output of the executable unit test program will be in another subdirectory of this directory (all of this layering can be annoying in cases like this, but it makes it possible to run large suites of regression tests without relying on any shared resources except for raw disk space). Note that the -z argument in the default test harness configuration tells the test harness to delete any prior instance of these results when you rerun the same test. If you want to preserve a history of results you can remove the -z and -i options from the runtests command and instead of using the statically named tmp directory it will create a unique timestamped subdirectory for each test run.

Simulation

In order to test and verify the control system in a workstation environment we need to be able to simulate the physical system under control. To provide the most accurate simulation, and to simplify the job of porting the result to the target embedded system, it is important that the control system interact with the simulation only through software

interfaces that will be common to the real physical system. In this case we do that by defining the public interfaces of our hardware adapters as the primary interfaces we want to preserve. Calls to the simulation are implemented in the hardware adapter. Note that we could also have chosen a lower-level device driver interface as the cross-platform standard here. Had we done that, we could have made the entire hardware adapter portable. However, since these hardware adapters are only interface wrappers themselves, this choice is expected to have little effect on the effort needed to port this to the physical target.

The simulation itself is encapsulated in a Simulation class that keeps track of the simulation states for the switches, sensor, and temperature using the models defined in the requirements. Note that the model for temperature as a function of time is encapsulated in its own separate class (IdealTempModel) so that the same model can be used in the simulation and in the switch health estimator.

System Tests

The requirements for the system tests are specified in the example state analysis document. The tests are implemented as follows:

<i>test</i>	<i>Make target</i>	<i>description</i>
1	thm_test1	Schedule and execute a knowledge activity over a background knowledge activity. Verify that all states are estimated.
2		Not implemented
3		Not implemented
4	thm_test4	Schedule and execute an activity to maintain the WARM temperature state.
5	thm_test5	Same as test 4, but simulate switch 1 failing stuck open as an initial condition (implementation note: initialize state estimate to this condition). Verify that tactic 1 schedules and runs.
6	thm_test6	Same as test 4, but simulate switch 1 failing stuck closed as an initial condition (implementation note: initialize state estimate to this condition) Verify that tactic 1 schedules and runs.
7	thm_test7	Same as test 4, but simulate switch 1 failing open during execution and verify goal failure response: re-elaborate tactic 2
8	thm_test8	Same as test 4, but simulate switch 2 failing closed during execution and verify goal failure response: re-elaborate tactic 2
9	thm_test9	Same as test 4, but simulate a sensor failure during execution and verify goal failure response (safing)
10	sgraph	Example graphic translation of goal network outputs (see verification/Helpers/MDS/GoalNet.pm for a description of the dependencies)

Table 7.: System Test Descriptions

Let's look at the results of the last test (9) as an example of how we verify the test success criteria.

The thm_test9 target in the root makefile executes under the test harness a test script which can be found in verification/Mds/Ra/SimpleThermostat/SystemTests/STest9.pl. This test script has two parts. The first part compiles and executes a unit test executable in the source directory source/Mds/Ra/SimpleThermostat/Tests. The second part of this test evaluates events reported in the event log file to verify some details of the execution that are difficult to evaluate during the execution. Specifically, it verifies that the goal network that was scheduled and executed ordered the time points in the correct order.

When you run the test from the MDS_ROOT directory using the root Makefile (make thm_test9) some summary results will be output directly to stdout; all of the details are

saved to test-specific output files. By default, the test results go into test-specific subdirectories of a results directory (named “results”) in the MDS_ROOT directory. The output from this test will be in the “results/tmp.thm_stest9” subdirectory. That directory contains several files used by the test harness including a file with the suffix “rslt” containing a more detailed summary of the results for this specific test, and a subdirectory (with suffix “dir”) containing the detailed results. In the detailed results directory you'll find a stdout file containing any compiler results, and the test harness output from launching the test executable. The actual output from the executable will be in a subdirectory given the same name as the executable.

Let's first look at the results reported in results/tmp.thm_stest9/Mds_Ra_SimpleThermostat_SystemTest_STest9.dir/stdout. The first line in this file reports which test script is being executed:

```
COMMAND:  
[...]/verification/Mds/Ra/SimpleThermostat/SystemTest/STest  
9.pl
```

Following that is a lot of compiler commands and resulting compiler output from compiling the test application, and then a couple of notes reporting that it is executing the Stest9 application. The last note reports that the application finished and returned a zero exit status:

```
NOTE: Mds_Ra_SimpleThermostat_SystemTest_STest9 [...]
returned zero (success) exit status
```

This indicates that the application ran and no CppUnit assertions were violated, so all of the internal tests passed. This particular test script uses a test helper to evaluate the order of goal scheduling and execution events that occur during the test. The test expects to schedule a particular goal network with time points scheduled in a particular order. When the goal network executes the test injects a sensor fault at a particular time which should cause the parent goal of this network to fail. So the test asserts that the last time point of planned network does not, in fact, fire, and that safing was invoked.

More detailed results can be found in the Stest9 subdirectory. Here you will find two key output files. Stest9.stdouterr contains all of the stdout and stderr output streams that the Stest9 application produced when it ran. Elf.log contains all of the formatted event log messages. Virtually everything in the stdouterr file is debug output, and there's quite a lot of it when the system is built in the default debug configuration. This includes initialization and finalization messages, details of the scheduling and execution process, and details of input parameters and output results from key methods in the adaptation.

An additional test, sgraph, is provided to demonstrate a primitive graphical transformation that can be performed on the goal network that these tests create. The actual test script for this test can be found at verification/Mds/Ra/SimpleThermostat/SystemTests/GraphicsTests.pl. This test uses a special test helper perl module that uses the GraphViz graphics library to produce graphical depictions. This requires some extra libraries be installed as described in the helper module (found at verification/Helpers/MDS/GoalNet.pm). Illustration 4 shows the goal elaboration graph produced by Stest4.pl (the full resolution version of this graphic can be found on the distribution disk at docs/example/goalnetElaboration.png).

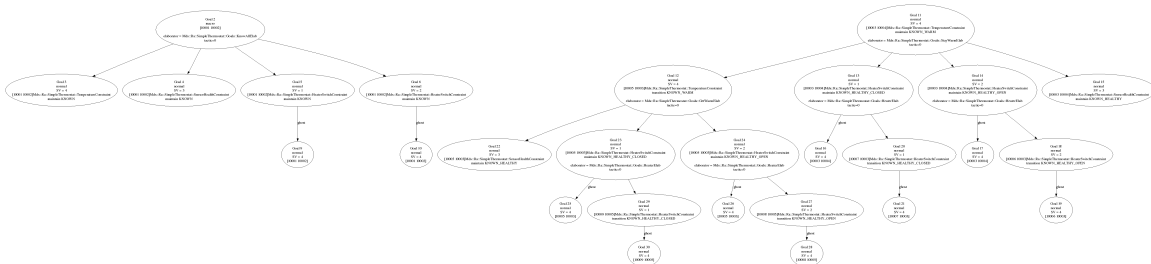


Illustration 4: Example Goalnet Elaboration from test 4

Appendix B. Step-By-Step Process Used In the Adaptations

This section provides a step-by-step description of the process used to develop the example control system. This provides more detail that is given in the guideline, and may be helpful to developers working through the process for the first time. Some of the steps described here may not apply to your process. Some of the steps described here are the result of a compressed development schedule in which requirements were being developed in parallel with the coding. This is not the recommended process for obvious reasons.

- 1) Create a new subdirectory SimpleThermostat in \$MDS_ROOT/source/Mds/Ra.
- 2) Create a make.cfg file for the new package.
- 3) Write measurement class
 - Copy \$MDS_ROOT/source/Mds/Ra/Example/Measurement.h and \$MDS_ROOT/source/Mds/Ra/Example/Measurement.cpp to SimpleThermostat/TemperatureMsmt.h and SimpleThermostat/TemperatureMsmt.cpp, respectively.
 - Add new ID to \$MDS_ROOT/source/Mds/Fw/Sid/Database.sid for the newly created measurement class TemperatureMsmt.
- 4) Flesh out TemperatureMsmt class
 - Define payload data as an unsigned char, per state analysis requirements.
 - Need to add an accessor for this member.
 - Need to augment constructors to initialize this new member.
- 5) Re-run metamake and build the core system. Verified code compiles with new adaptation code in place.
- 6) Write unit tests for the measurement class.
 - Add new subdirectory Test/ under SimpleThermostat

- Add common test code, and unit test cases to test all methods in measurement class.
- Add new scripts to run these unit tests under
\$MDS_ROOT/verification/Mds/Ra/SimpleThermostat/Measurement/
- Add new make targets for the unit tests in top-level makefile under \$MDS_ROOT

7) Verify all unit tests pass.

8) Write command class

- Copy \$MDS_ROOT/source/Mds/Ra/Example/Command.h and
\$MDS_ROOT/source/Mds/Ra/Example/Command.cpp to
SimpleThermostat/HeaterCmd.h and SimpleThermostat/HeaterCmd.cpp,
respectively.
- Add new ID to \$MDS_ROOT/source/Mds/Fw/Sid/Database.sid for the newly
created measurement class HeaterCmd.
- Define desired switch state parameter as an unsigned char, per State Analysis
requirements.
- Need to add an accessor for this member.
- Need to augment constructors to initialize this new member.
- Re-run metamake and build the core system. Verified code compiles with new
adaptation code in place.

9) Write unit tests for the command class.

- Add new subdirectory Test/ under SimpleThermostat/
- Add common test code, and unit test cases to test all methods in command class.
- Add new scripts to run these unit tests under
\$MDS_ROOT/verification/Mds/Ra/SimpleThermostat/Command/
- Add new make targets for the unit tests in top-level makefile under \$MDS_ROOT

10) Verify all unit tests pass.

11) Write state value class

- Copy `$MDS_ROOT/source/Mds/Ra/Example/StateVal.h` and `$MDS_ROOT/source/Mds/Ra/Example/StateVal.cpp` to `SimpleThermostat/HeaterSwitchStateVal.h` and `SimpleThermostat/HeaterSwitchStateVal.cpp`, respectively.
- Add new ID to `$MDS_ROOT/source/Mds/Fw/Sid/Database.sid` for the newly created measurement class `HeaterSwitchStateVal`.
- Define enumeration for state values (as defined during state analysis).
- Create one enumeration to handle operational mode, knowledge state, and health state.
- Need to add an accessor for this member.
- Need to augment constructors to initialize this new member.
- Re-run `metamake` and build the core system. Verified code compiles with new adaptation code in place.

12) Write unit tests for the state value class.

- Add new subdirectory `Test/` under `SimpleThermostat/`
- Add common test code, and unit test cases to test all methods in state value class.
- Add new scripts to run these unit tests under `$MDS_ROOT/verification/Mds/Ra/SimpleThermostat/StateValue/`
- Add new make targets for the unit tests in top-level makefile under `$MDS_ROOT`

13) Verify all unit tests pass.

14) Write state function class for switch state.

- For switch state function, simply use the `ConstantStateFunction`.
- Add new ID to `$MDS_ROOT/source/Mds/Fw/Sid/Database.sid` for the newly created measurement class `HeaterSwitchStateVal`.
- Define traits struct and state function in new header `HeaterSwitchStateFunction.h`

15) In the interest of time, defer writing unit tests for the state function and value history at the present time as we're just using a pre-canned class.

16) Write the state variable class for the heater switch.

- Create new SV class, HeaterSwitchSV.
- Implement ctor, dtor, and both getState() methods.
- Add skeletons for isStillSatisfiable() methods, without constraints.
- Comment out getUnconstrainedConstraint() until constraint class is written.

17) Write skeleton versions of the following classes:

- Switch constraint
- Switch estimator
- Switch controller

18) Write hardware adapter for heater switch.

19) Write separate simulation component to perform temperature simulation as well as provide interfaces for fault injection.

20) Write unit tests for temperature simulation model as well as sensor measurement interface. Verify that model performs as specified in the requirements.

21) Write hardware adapter for temperature sensor.

22) At this point, all components for the heater switch control diamond are coded. It makes sense to start thinking about hooking these up and getting them to run. However, in order to do this, we require at least compilable skeletons for the other components. The next step is write skeletons for the remaining classes and provide default implementations for most methods (for instance, MPE methods just return true):

- Sensor Health SV
- Sensor Health Estimator
- Sensor Health Constraint
- Sensor Health State Value
- Sensor Health State Function
- Temperature SV
- Temperature Estimator
- Temperature Constraint

- Temperature State Value
- Temperature State Function

23) Since all of the state value classes have a lot of similarities, write unit tests for all state value classes, using the test for the heater switch state value as a template.

24) In order to run the simplest test, we need to be able to schedule a unconstrained network. This requires that the state constraint classes be well-defined but not fully implemented. Now, define the constraint classes and implement all of the methods on them *except* for the MPE methods (just make them return true for now).

- As with the state values, the state constraint classes are all similar in structure. It makes sense to write some unit tests for the state constraints now, deferring tests for merge and subset methods until they are implemented later.
- For this adaptation, the specifications for the MPE methods came later than expected. Ideally, one would like to write the entire constraint class, including subset and merge methods, and spend some time to write unit tests to verify proper behavior of these methods early on and not have to worry about them later.

25) At this point, all classes have at least skeletons. Add a new subdirectory, System/, and define a new class ThermostatArch to aggregate all achitectural elements and hook them up properly.

26) Create two classes, SimpleThermostatUtestCommon (for unit tests) and SimpleThermostatStestCommon (for system tests) using the existing SimpleThermostatTestCommon as a base. This allows for different common test code for unit and system tests.

27) Instantiate and properly initialize architecture aggregate object in common system test code.

28) Create a basic system test which simply instantiates the system, schedules and executes the unconstrained network for several minutes. Verify that this test compiles and runs with no major anomalies (seg faults, etc).

29) The big tasks left at this point are: coding the achiever algorithms and the MPE methods. Since we don't yet have the specifications for the MPE methods, write the achiever algorithms.

- HeaterSwitchController::control (switch controller algorithm)

- SensorHealthEstimator::estimate (sensor health estimation algorithm)
- TemperatureEstimator::estimate (temperature estimation algorithm)
- HeaterSwitchEstimator::estimate (switch estimation algorithm)
- The switch estimation algorithm is, by far, some of the most complex logic in this example. The algorithm laid out by Mitch was a good start but needed a bit more by way of details. Ken is going to spend some time working out the details of the estimation algorithm, so this one will need to be revisited later on.

30) Start coding MPE methods.

- Start with MPE methods for the sensor health, as they are the simplest. Per the Adapters Guide, code the schedule-time MPE methods first, followed by the execution-time methods. That looks like:
- SensorHealthSV::projectSerially
- SensorHealthConstraint::isSubsetOf
- SensorHealthSV::isAchievable
- SensorHealthSV::isTransitionAchievable
- SensorHealthSV::isReadyToTransition
- SensorHealthSV::isStillSatisfiable
- Here we identified a good optimization: the logic in isStillSatisfiable for both Goals and Xgoals should be the same. Therefore, it makes sense to have a private isStillSatisfiable method on each SV which performs the satisfiability checks, and have the public isStillSatisfiable methods call the private method. It might also be nice to put this in the base class and only have the private isStillSatisfiable be adapted.

31) As MPE methods are being developed, it is essential to have system tests ready to run which will be used to assess behavior of the system. Dave is working on the tests, while I continue to fight with these MPE methods. Is there possibly a better way to represent these requirements?

32) Refine switch estimation algorithm with Ken. Several iterations are needed before a final version is settled on.

33) Finish MPE methods for sensor health. Tests are running as of this point. Now move on to switch MPE methods, since they are similar (no affecting state) but slightly more complicated than the sensor health MPE methods.

34) Finish switch MPE methods, and Dave finishes temperature MPE methods (but only a limited set of functionality corresponding to the cases that will be exercised by our tests.)

34) All code written, including some late breaking updates to the switch estimation algorithm.

35) Test test test!

Bibliography

- 1 Dvorak, D., Rasmussen, R., Reeves, G., Sacks, A.; *Software Architecture Themes in JPL's Mission Data System*; IEEE Aerospace Conference, 2000
- 2 Rasmussen, R.; *Goal-Based Fault Tolerance for Space Systems Using the Mission Data System*; IEEE Aerospace Conference, 2001
- 3 Bennet, M., Rasmussen, R.; *Modelling Relationships Using Graph State Variables*; IEEE Aerospace Conference 2002
- 4 Dvorak, D., Rasmussen, R., Starbird, T.; *State Knowledge Representation in the Mission Data System*; IEEE Aerospace Conference 2002
- 5 Dvorak, D.; *Challenging Encapsulation in the Design of High-Risk Control Systems*; OOPSLA 2002
- 6 Ingham, M., Rasmussen, R., Bennett, M., Moncada, A.; *Engineering Complex Embedded Systems with State Analysis and the Mission Da*; AIAA Intelligent Systems Conference 2004
- 7 Ingham, M., Rasmussen, R., Bennett, M., Moncada, A.; *Generating Requirements for Complex Spacecraft Systems using State Analysis*; International Astronautical Federation Congress 20
- 8 Bennett, M., Knight, R., Rasmussen, R., Ingham, .M.; *State-Based Models for Planning and Execution*; International Conference on Automated Planning & S
- 9 Barrett, A., Knight, R., Morris, J.R., Rasmussen, R.; *Mission Planning and Execution Within the Mission Data System*; Proceedings of the International Workshop on Planning and Scheduling for Space 2004
- 10 Rasmussen, R., Ingham, M., Dvorak, D.; *Achieving Control and Interoperability Through Unified Model-Based Engineer*; AIAA Infotech and Aerospace Conference 2005
- 11 Wagner, D.; *Data Management in the Mission Data System*; IEEE System, Man, and Cybernetics Conference 2005
- 12 Szyperski, Clemens; *Component Software*; Addison-Wesley; 2002; ISBN 0201745720

Glossary

<i>Term</i>	<i>Definition</i>	<i>Traditionally...</i>
Achiever	An estimator or a controller.	See estimator and controller.
Allocation	The part of a delegation goal that specifies the bounds of allowable control.	
Basis state variable	A state variable estimated in a deployment.	Global variable.
Bonus goals	Low priority goal added to the plan if projected resources allow	“Bonus” sequence activity.
Command	Time-tagged outgoing directive to change the condition of one or more physical states.	Command
Command model	Describes instantaneous effects of a command on one or more physical states.	Model used by fault protection, sequencing, or analysis tools.
Control goal	A goal on the value of an estimated state of a state variable.	Flight rule.
Control system	Has cognizance over the system under control. State variables, estimators, controllers, planner, execution engine, goal networks.	Sequencing, Estimation & Control, Fault Protection, etc ...
Controller	Controls the physical state represented by one or more state variables. Achieves control goals.	Guidance, navigation, and control logic.
Data command	Outgoing directive to change the condition or transport of one or more value histories.	Downlink priorities.
Data controller	Controls one or more data state variables.	Telemetry manager.
Data state variable	Represents the state of one or more value histories.	Data catalogs.
Deployment	A partition of the control system into a physically separate location.	Flight system, testbed, simulation.

<i>Term</i>	<i>Definition</i>	<i>Traditionally...</i>
Delegate achiever	An achiever that sends goals to a delegating achiever.	On-board controller.
Delegating achiever	The achiever receiving goals from a delegate achiever.	On-board guidance and control commander.
Delegation	Achiever sends goals directly to other achiever during execution.	Use of on-board commander.
Derived state variable	Represents a relationship between multiple state variables.	Power margin.
Distillation	Estimator conversion of a measurement into an idealized distilled measurement.	On-board measurement filtering.
Elaboration	Specifies a block of supporting goals on state variables that reflect a state effects model and can be assembled into plans.	Sequence expansion or macro.
Estimated state	The control system's knowledge about a physical state.	Onboard variable updated by fault diagnosis; guidance, navigation, and control; mobility; event-based sequence engine, etc ...
Estimator	Updates estimated state for one or more state variables. Achieves knowledge goals.	Fault diagnosis. Calibration ground software.
Executable goal (Xgoal)	Element of a scheduled plan to be executed. Contains merged ordinary and delegation goals over an interval and the projection for that interval.	Closed-loop command.
Goal	A state constraint. A condition for a state variable that persists over an interval of time. Represents operator intent. Expressed as an assertion.	Closed-loop command. Flight rule.

<i>Term</i>	<i>Definition</i>	<i>Traditionally...</i>
Goal failure	State constraint violation caught during execution.	On-board pointing constraint violation, etc...
Goal network	Collection of interconnected goals and time points resulting from elaborations	Parallel sequences.
Hardware adapter	Provides a measurement and command interface between the hardware of the system under control hardware and the control system. Keeps one or command and measurement value histories.	Device managers. Device drivers.
Intended state	The control system's prediction for a state variable based on the operator intent as scheduled in the plan.	Sequence predicts without use of models and initial conditions.
Is achievable	Defined for both goals and Xgoals. A goal is achievable if is within the capability of its achiever. An Xgoal is achievable if its elements are consistent.	Sequence and command constraint checking.
Is still satisfiable	Evaluation of a goal over its interval of time	Fault monitor.
Is transition achievable	Achiever capability to successfully execute two Xgoals back-to-back.	Sequence checking and command constraint checking.
Knowledge goal	A goal on the quality of estimated state knowledge for a state variable.	Guidance, navigation, and control mode commands.
Macro goal	Expands to goal net on state variables that may be unrelated in the State Effects Model.	Sequence expanded block.
Measurement	Provides time-tagged evidence about one or more physical states for a moment in time. May be a science observation.	Telemetry. Science observation. Image.
Measurement model	Describes how one or more physical states affect a sensor's measurement.	Includes telemetry calibration parameters.
Merging	The combination of concurrent goals over a time interval into a single goal that fully satisfies the intent of all concurrent goals.	Sequence integration and checking.

<i>Term</i>	<i>Definition</i>	<i>Traditionally...</i>
Model ("The Model")	All state variables, state effects models, command models, and measurement models.	See state variable, state effects model, command model, measurement model.
Physical state	Exists in the system under control. May be a hardware state, environment state, or even a software state.	May appear in models used by fault protection, sequencing, or analysis tools.
Projected state	The control system's prediction for a state variable based on latest estimated state, operator intent as scheduled in the plan, the model, and achiever behavior.	Prediction of sequence execution, resource usage (e.g., power/energy, data storage), side effects (e.g., thermal, interference), and characterization of software behavior (e.g., pointing, mobility).
Proxy state variable	A copy of basis state variable whose value is estimated in a separate deployment.	Uplinked calibration parameter. Software telemetry.
Reachable state	The control system's prediction for a state variable based its the latest estimated state and what is possible given the model.	Fault diagnosis predictions. Ephemeris predictions.
Ready to transition	Specified conditions under which one Xgoal can stop executing and the next one begin.	Event-based sequencing.
Safe goal net	A goal network that establishes a safe system state.	Safing fault reponse.
Safing	The process of bringing the system to a safe (recoverable) state after fault has been detected.	Same

<i>Term</i>	<i>Definition</i>	<i>Traditionally...</i>
Scheduling	1. Merging of elaborated goals into the current goal network, scheduling goals to satisfy all temporal constraints and eliminate any conflicts, and verifying that the resulting plan is consistent with projected state predictions. Performed after elaboration. 2. Low-level process of determining what instructions should run on the processor at any given time.	1. Sequence generation and checking (e.g., SEQGEN modeling). 2. Same
State effects model	Describes the behavior of a physical state including how other physical states affect it.	Model used by fault protection, sequencing, or analysis tools.
State function	Describes the change of an estimated state over an interval of time. Produces state values	Trajectory function
State value	Estimated state for a moment in time. Includes uncertainty.	Global variable value.
State variable	Represents a property of a thing in the system under control.	Global variable.
System under control	The vehicle, its environment, and certain software elements such as hardware I/O and data management and transport functions.	The same.
Tactic	Alternate way of elaborating a goal - produces a different goal network when certain conditions are met.	Conditional sequence expansion.
Temporal Constraint	Timing requirement between 2 time points. Can specify order, min time, max time, or a range of time.	Sequence command time (absolute, relative).
Time line	Expresses a state variable's estimated, intended, projected, or reachable state.	Sequence predicts.
Time point	Represents a moment in time in a plan (goal network).	Command time in a sequence.
Value history	Contains state functions, commands, or measurements. Resides in the system under control. Content can be controlled with data commands.	Telemetry or command buffer.

<i>Term</i>	<i>Definition</i>	<i>Traditionally...</i>
XGoal	See executable goal.	

Index

A

Achiever.....	
Achiever.....	6
achievers.....	11, 12, 24, 32, 36, 48, 63
Achievers.....	9, 11, 31, 36, 42, 60

C

Commands.....	26
component software architecture.....	41
Controller.....	6, 36

D

data goals.....	39
data management.....	39

E

elaboration.....	11, 12, 29, 38, 54, 59, 85, 87
Elaboration.....	8, 9, 28, 38, 41, 54, 67, 84
elaborator.....	6, 9, 12, 24, 29, 59
Elaborator.....	8, 11, 24, 30, 37, 38, 45, 62
Elf Events.....	43
Estimator.....	6, 36
event generator.....	43
event message.....	43
executable	8
executable component.....	41
Execution.....	10, 41

G

Goal.....	7
goal network.....	8
Goal Network.....	7

H

Hardware Adapter.....	6, 37
-----------------------	-------

I

initialization.....	44
Initialization.....	45

M

macro goal.....	37
Measurements.....	25
metamake.....	16, 19, 23, 52

Metamake.....	16, 17, 18, 19, 20
MPE.....	12
N	
namespace.....	15
O	
optimization.....	27
Optimization.....	46, 48
ordinary constraint.....	9
P	
Planning and Execution Methods.....	32
projected constraint.....	9
Projection.....	9
Promotion.....	9
Q	
Query Functors.....	29
R	
RateGroup.....	41
realtime.....	42
S	
safing.....	11
Safing.....	11
Scheduling.....	
Goal Scheduling.....	
scheduling.....	8, 24, 34, 36, 37, 41, 54, 63, 66, 73, 74, 87
Scheduling.....	8, 13, 41, 54, 87
simulation.....	24, 37, 55
Simulation.....	55
State Analysis Artifacts.....	15
state constraint.....	6, 7, 9, 13, 20, 24, 29, 30, 33, 34, 59, 61, 64, 66, 67, 68, 79, 84
state Constraint.....	29, 36
State constraint.....	29, 34, 61, 66, 85
State Constraint.....	7, 29, 66
State Functions.....	26
State Values.....	26
State Variable.....	6, 31
System ID Database.....	20
T	
Tactic.....	8
Tactics.....	24, 37, 38
telemetry.....	43

temporal constraint.....	10, 39, 40, 54, 87
Temporal Constraint.....	7, 8, 11, 87
terminal goal.....	37
Testing.....	53
thread.....	41
Threading.....	56
Time Point.....	
timepoint.....	
time point.....	10, 29, 38, 39, 40, 53, 55, 62, 72, 73, 87
Time point.....	87
Time Point.....	7, 8, 9, 10, 11
U	
unconstrained.....	30
Update Functors.....	29
V	
Value Histories.....	28
Value History.....	
value history.....	28, 29, 31, 32, 39, 45, 47, 48, 56, 65, 66, 77
Value history.....	28, 48, 87
Value History.....	7, 56
X	
XGoal.....	8